普通高等教育“十一五”国家级规划教材

谭浩强 主编

高职高专计算机教学改革新体系规划教材

ARM 9嵌入式系统设计与应用

李新荣 曲凤娟 编著

清华大学出版社

VISIT…

LANZAROTE Caliente.COM

普通高等教育“十一五”国家级规划教材
高职高专计算机教学改革新体系规划教材

ARM 9 嵌入式系统设计与应用

李新荣　曲凤娟　编著

清华大学出版社
北　京

内 容 简 介

本书以 ARM 9 处理器和 Linux 操作系统为平台，从 S3C2440A 处理器及其外围设备的基本知识讲起，然后介绍在 Linux 操作系统下开发嵌入式系统所需的知识，最后介绍开发一个典型的嵌入式系统的全过程，使学生对嵌入式系统的开发有一个全面的认识，为今后从事嵌入式系统开发奠定了基础。

本书深入浅出，适合计算机及相关专业的师生使用，也可作为嵌入式系统爱好者学习嵌入式系统设计的入门教材和嵌入式系统开发人员的技术参考书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目（CIP）数据

ARM 9 嵌入式系统设计与应用/李新荣，曲凤娟编著．—北京：清华大学出版社，2011.6
（高职高专计算机教学改革新体系规划教材）
ISBN 978-7-302-25340-2

Ⅰ．①A… Ⅱ．①李… ②曲… Ⅲ．①微处理器—系统—设计—高等职业教育—教材
Ⅳ．①TP332

中国版本图书馆 CIP 数据核字(2011)第 068720 号

责任编辑：张 景
责任校对：刘 静
责任印制：何 芊

出版发行：清华大学出版社 **地 址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn **邮 编**：100084
社 总 机：010-62770175 **邮 购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市春园印刷有限公司
经 销：全国新华书店
开 本：185×260 **印 张**：20.75 **字 数**：498 千字
版 次：2011 年 6 月第 1 版 **印 次**：2011 年 6 月第 1 次印刷
印 数：1～3000
定 价：39.00 元

产品编号：031467-01

丛书编委会

主　任　谭浩强

副主任　丁桂芝　李凤霞　焦金生

委　员　孔令德　王天华　王兴玲　王学卿
刘　星　安淑芝　安志远　宋金珂
宋文官　沈　洪　束传政　邵丽萍
尚晓航　张　玲　张翰韬　林小茶
赵丰年　高文胜　秦建中　崔武子
谢　琛　薛淑斌　熊发涯

序

近年来，我国高等职业教育迅猛发展，目前，高等职业院校已占全国高等学校半数以上，高职学生数已超过全国大学生的半数。高职教育已占了我国高等教育的“半壁江山”。发展高职教育，培养大量技术型和技能型人才，是国民经济发展的迫切需要，是高等教育大众化的要求，是促进社会就业的有效措施，也是国际教育发展的趋势。

高等职业教育是我国高等教育的重要组成部分，高职教育的质量直接影响了全国高等教育的质量。办好高职教育，提高高职教育的质量已成为我国教育事业中的一件大事，已引起了全社会的关注。

为了更好地发展高职教育，首先应当建立起对高职教育的正确理念。

高职教育是不同于普通高等教育的一种教育类型。它的培养目标、教学理念、课程体系、教学内容和教学方法都与传统的本科教育有很大的不同。高职教育不是通才教育，而是按照职业的需要，进行有针对性培养的教育，是以就业为导向，以职业岗位要求为依据的教育。高职教育是直接面向市场、服务产业、促进就业的教育，是高等教育体系中与经济社会发展联系最密切的部分。

在高职教育中要牢固树立“人才职业化”的思想，要最大限度地满足职业的要求。衡量高职学生质量的标准，不是看学了多少理论知识，而是看会做什么，能否满足职业岗位的要求。本科教育是以知识为本位，而高职教育是以能力为本位的。

强调以能力为本位，并不是不要学习理论知识，能力是以知识为支撑的，问题是学什么理论知识和怎样学习理论知识。有两种学习理论知识的模式：一种是“建筑”模式，即“金字塔”模式，先系统学习理论知识，打下宽厚的理论基础，以后再结合专业应用；另一种是“生物”模式，如同植物的根部、树干和树冠是同步生长的一样，随着应用的开展，结合应用学习必要的理论知识。对于高职教育来说，不应该采用“金字塔”模式，而应当采用“生物”模式。

可以比较一下以知识为本位的学科教育和以能力为本位的高职教育在教学各个方面的不同。知识本位着重学习一般科学技术知识；注重的是系统的理论知识，讲求的是理论的系统性和严密性；学习要求是“了解、理解、掌握”；构建课程体系时采用“建筑”模式；教学方法采用“提出概念—解释概念—举例说明”的传统三部曲；注重培养抽象思维能力。而能力本位着重学习工作过程知识；注重的是实际的工作能力，讲求的是应用的熟练性；学习要求是“能干什么，达到什么熟练程度”；构建课程体系时采用“生物”模式；教学方法采用“提出问

题—解决问题—归纳分析”的新三部曲;常使用形象思维方法。

近年来,国内教育界对高职教育从理论到实践开展了深入的研究,引进了发达国家职业教育的理念和行之有效的做法,许多高职院校从多年的实践中总结了成功的经验,有力地推动了我国的高职教育。再经过一段时期的研究与探索,会逐步形成具有中国特色的完善的高职教育体系。

全国高校计算机基础教育研究会于 2007 年 7 月发布了《中国高职院校计算机教育课程体系 2007》(以下简称《CVC 2007》),系统阐述了高职教育的指导思想,深入分析了我国高职教育的现状和存在的问题,明确提出了构建高职计算机课程体系的方法,具体提供了各类专业进行计算机教育的课程体系参考方案,并深刻指出为了更好地开展高职计算机教育应当解决好的一些问题。《CVC 2007》是一个指导我国高职计算机教育的重要的指导性文件,建议从事高职计算机教育的教师认真学习。

《CVC 2007》提出了高职计算机教育的基本理念是:面向职业需要、强化实践环节、变革培养方式、采用多种模式、启发自主学习、培养创新精神、树立团队意识。这是完全正确的。

教材是培养目标和教学思想的具体体现。要实现高职的教学目标,必须有一批符合高职特点的教材。高职教材与传统的本科教育的教材有很大的不同,传统的教材是先理论后实际,先抽象后具体,先一般后个别,而高职教材则应是从实际到理论,从具体到抽象,从个别到一般。教材应当体现职业岗位的要求,紧密结合生产实际,着眼于培养应用计算机的实际能力。要引导学生多实践,通过“做”而不是通过“听”来学习。

评价高职教材的标准不是愈深愈好、愈全愈好,而是看它是否符合高职特点,是否有利于实现高职的培养目标。好的教材应当是“定位准确,内容先进,取舍合理,体系得当,风格优良”。

教材建设应当提倡百花齐放,推陈出新。我国高职院校为数众多,情况各异。地域不同、基础不同、条件不同、师资不同、要求不同,显然不能一刀切,用一个大纲、一种教材包打天下。应该针对不同的情况,组织编写出不同的教材,供各校选用。能有效提高教学质量的就是好教材。同时应当看到,高职计算机教育发展很快,新的经验层出不穷,需要加强交流,推陈出新。

从 20 世纪 90 年代开始,我们开始注意研究高职教育,并在 1999 年组织编写了一套“高职高专计算机教育系列教材”,由清华大学出版社出版,这是在国内最早出版的高职教材之一。在国内产生很大的影响,被许多高职院校采用为教材,有力地推动了蓬勃兴起的高职教育,后来该丛书扩展为“高等院校计算机应用技术规划教材”,除了高职院校采用之外,还被许多应用型本科院校使用。几年来已经累计发行近 300 万册,被教育部确定为“普通高等教育‘十一五’国家级规划教材”。

根据高职教育发展的新形势,我们于 2005 年开始策划,在原有基础上重新组织编写一套全新的高职教材——“高职高专计算机教学改革新体系规划教材”,经过两年的研讨和编写,于 2007 年正式由清华大学出版社出版。这套教材遵循高职教育的特点,不是根据学科的原则确定课程体系,而是根据实际应用的需要组织课程;书名不是按照学科的角度来确定的,而是体现应用的特点;写法上不是从理论入手,而是从实际问题入手,提出问题、解决问题、归纳分析、循序渐进、深入浅出、易于学习,有利于培养应用能力。丛书的作者大都是多

年从事高职院校计算机教育的教师，他们对高职教育有较深入的研究，对高职计算机教育有丰富的经验，所写的教材针对性强，适用性广，符合当前大多数高职院校的实际需要。这套教材经教育部审查，已列入“普通高等教育‘十一五’国家级规划教材”。

本套教材统一规划，分工编写，陆续出版，逐步完善。随着高职教育的发展将会不断更新，与时俱进。恳切希望广大师生在使用中发现本丛书不足之处，并不吝指正，以便我们及时修改完善，更好地满足高职教学的需要。

全国高校计算机基础教育研究会 会长
“高职高专计算机教学改革新体系规划教材”主编 谭浩强

前言

基于ARM技术的微处理器应用占据了32位RISC微处理器的大部分市场，并渗入到人们生活的各个方面。本书将以基于ARM920T核的S3C2440A芯片为核心，以Linux操作系统为平台，详细介绍嵌入式系统的设计与开发过程、调试方法。

本书共分8章，每章内容介绍如下：

第1章　嵌入式系统概述　介绍目前嵌入式系统的发展状况；嵌入式系统的定义、特点、应用领域；嵌入式系统的组成，包括嵌入式硬件组成与软件系统，为读者以后的学习打下基础。

第2章　ARM微处理器　介绍ARM处理器的技术特点和应用领域；ARM处理器体系结构、版本、变种及版本命名格式；ARM处理器的编程模型，包括ARM处理器的数据类型、工作状态、工作模式和寄存器组织、存储模式、I/O端口的访问方式和异常；ARM微处理器的选型。

第3章　ARM程序设计基础　重点讲述ARM汇编程序设计基本编程方法，包括ARM及Thumb指令集以及ARM寻址方式，并通过范例进一步讲解ARM指令集的使用方法；ARM汇编语言和汇编程序规范、程序格式；ARM汇编器的伪操作和伪指令的使用；ARM汇编程序中的常用符号、表达式、运算符和程序的基本结构；最后结合实例介绍汇编语言与C/C++语言的混合编程。

第4章　嵌入式系统硬件设计　首先介绍嵌入式最小系统的设计和S3C2440A芯片设计，然后重点介绍S3C2440A外围部件的工作原理，包括存储器控制器、Nand Flash控制器、中断控制器、通用I/O口、串行通信和定时器；嵌入式系统硬件基本电路，包括电源、复位、晶振电路、存储器接口和JTAG接口、串行接口等；S3C2440A启动程序，包括中断向量表、初始化存储器系统、初始化堆栈、初始化有特殊要求的端口和设备、初始化用户程序执行环境、改变处理器模式、呼叫主应用程序。

第5章　嵌入式操作系统基础　详细介绍操作系统的基础知识，包括操作系统的定义、功能、特征、类型；进程和线程的基本知识；中断和中断的处理；内核的分类等基础知识，这些都是开发嵌入式系统必不可少的基础知识。本章还将对当今流行的开源和商业的嵌入式操作系统进行介绍，最后针对Linux系统的特点、组成和应用前景展开介绍。

第6章　嵌入式Linux开发基础　从Linux的基本知识、常用命令讲起，这是学习Linux的入门知识，然后通过实例讲述Linux C编程的基本过程及相

应的开发工具,包括 vi 和 emacs 编辑工具、gcc 编译工具、make 工程管理工具和 gdb 调试工具的使用,版本控制的基本概念,这些都是进行 Linux 开发必须掌握的工具。本章还将介绍多进程和多线程的开发,讲述 Linux 下进程和线程编程的基本方法,相应地介绍多进程和多线程的程序调试方法。最后讲解交叉编译的概念,通过实例分析如何将一个 Linux 的程序交叉编译为在 ARM 处理器上运行的程序。

第 7 章 构建嵌入式 Linux 系统 讲述构建 Linux 系统的全过程,包括嵌入式 Linux 的组成、开发主机和目标机之间的通信、Bootloader 的启动、Linux 内核的移植和配置、根文件系统的构建等内容。

第 8 章 基于 Web 的远程监控系统的设计实例 详细介绍一个基于 Web 的远程监控系统的设计过程,包括系统架构设计和软、硬件的实现。本章首先介绍嵌入式 Web 服务器和远程监控系统的概念,然后介绍嵌入式 Web 远程监控系统的整体架构设计,其中网络架构以嵌入式 Web 服务器为中心,通过 Internet 远程访问嵌入式 Web 服务器,嵌入式 Web 服务器通过现场总线控制各个节点,以达到远程监控的目的;硬件架构采用三星公司的主流 ARM 9 处理器 S3S2440A 进行构建;软件架构以 Linux 操作系统为平台,选择 boa+CGI 方案。

本书由李新荣负责编写第 3、5、6、7、8 章,并对本书进行了统稿,曲凤娟负责编写第 1、2、4 章。李建义、房好帅、王慧娟、金大兵、李楠、王静、刘立媛等也参加了本书大纲的讨论和部分内容的编写。

由于作者水平有限,书中不足之处在所难免,敬请广大读者批评指正,本书作者的邮箱为 l_xinrong@sina.com(李新荣)。

编 者

2011 年 5 月

目录

第 1 章　嵌入式系统概述 …… 1
1.1　嵌入式系统基础 …… 2
1.1.1　嵌入式系统的发展历史 …… 2
1.1.2　嵌入式系统的定义与特点 …… 4
1.1.3　嵌入式系统的组成 …… 5
1.1.4　嵌入式系统的应用领域 …… 6
1.1.5　嵌入式技术的发展趋势 …… 7
1.2　嵌入式系统的硬件组成 …… 8
1.2.1　嵌入式处理器 …… 9
1.2.2　嵌入式外围设备与接口 …… 10
1.2.3　典型的嵌入式处理器与开发板 …… 11
1.3　嵌入式系统的软件组成 …… 13
1.3.1　嵌入式软件的基本特点与分类 …… 13
1.3.2　嵌入式软件开发环境 …… 15
1.3.3　嵌入式软件开发的要点 …… 16
1.3.4　嵌入式操作系统 …… 18
小结 …… 19

第 2 章　ARM 微处理器 …… 20
2.1　ARM 微处理器概述 …… 21
2.1.1　ARM 微处理器的技术特点 …… 21
2.1.2　ARM 微处理器的应用领域 …… 22
2.2　ARM 微处理器体系结构 …… 22
2.2.1　RISC 体系结构 …… 22
2.2.2　ARM 体系结构版本 …… 23
2.2.3　ARM 体系结构的变种及版本命名格式 …… 25
2.2.4　ARM 微处理器系列 …… 26
2.3　ARM 微处理器的编程模型 …… 30
2.3.1　ARM 微处理器的数据类型 …… 30
2.3.2　ARM 微处理器的工作状态 …… 30
2.3.3　ARM 微处理器的工作模式 …… 31

2.3.4 ARM 微处理器的寄存器组织 …… 32
2.3.5 ARM 体系中的存储模式 …… 36
2.3.6 I/O 端口的访问方式 …… 36
2.3.7 异常 …… 37
2.4 ARM 微处理器的选型 …… 38
小结 …… 39

第 3 章 ARM 程序设计基础 …… 40
3.1 ARM 指令系统 …… 41
3.1.1 ARM 指令系统概述 …… 41
3.1.2 ARM 寻址方式 …… 43
3.1.3 ARM 指令集 …… 46
3.1.4 Thumb 指令集 …… 56
3.2 ARM 汇编语言和汇编语言编程规范 …… 58
3.2.1 ARM 汇编语言语句格式 …… 58
3.2.2 ARM 汇编器的伪操作 …… 59
3.2.3 ARM 汇编器支持的伪指令 …… 69
3.3 ARM 汇编语言程序格式 …… 71
3.3.1 ARM 汇编语言程序中常用的符号 …… 71
3.3.2 汇编语言程序中的表达式和运算符 …… 74
3.3.3 ARM 汇编语言程序的基本结构 …… 76
3.3.4 ARM 汇编程序设计举例 …… 77
3.4 汇编语言与 C/C++ 语言的混合编程 …… 80
3.4.1 在 C/C++ 程序中嵌入汇编指令 …… 80
3.4.2 在 ARM 汇编程序和 C/C++ 程序之间进行变量的互访 …… 81
3.4.3 汇编程序、C/C++ 程序间的相互调用 …… 82
小结 …… 84

第 4 章 嵌入式系统硬件设计 …… 85
4.1 嵌入式最小系统 …… 86
4.2 S3C2440A 概述 …… 87
4.3 S3C2440A 外围部件工作原理 …… 88
4.3.1 存储器控制器 …… 89
4.3.2 Nand Flash 控制器 …… 97
4.3.3 中断控制器 …… 105
4.3.4 通用 I/O 口 …… 121
4.3.5 串行通信 …… 130
4.3.6 定时器 …… 141
4.4 嵌入式系统硬件基本电路 …… 149
4.5 S3C2440A 启动程序 …… 152

小结…… 156

第 5 章 嵌入式操作系统基础…… 157

5.1 操作系统的基本概念 …… 158

5.1.1 操作系统的定义…… 158

5.1.2 操作系统的功能…… 159

5.1.3 操作系统的基本特征…… 161

5.1.4 进程和线程的基本概念…… 162

5.1.5 进程的同步与互斥…… 164

5.2 中断和中断处理 …… 165

5.2.1 中断…… 165

5.2.2 中断处理与中断返回…… 166

5.3 单内核与微内核 …… 167

5.3.1 内核…… 167

5.3.2 单内核操作系统与微内核操作系统…… 168

5.4 操作系统的类型 …… 169

5.4.1 单用户操作系统…… 169

5.4.2 批处理操作系统…… 170

5.4.3 分时操作系统…… 170

5.4.4 实时操作系统…… 171

5.5 当今流行的嵌入式操作系统简介 …… 172

5.5.1 嵌入式操作系统的发展…… 172

5.5.2 使用嵌入式操作系统的必要性…… 173

5.5.3 嵌入式操作系统选型…… 174

5.5.4 常见的开源嵌入式操作系统简介…… 175

5.5.5 常见的商业嵌入式操作系统简介…… 176

5.6 Linux 系统简介 …… 178

5.6.1 Linux 的特性 …… 178

5.6.2 Linux 版本及其特点 …… 180

5.6.3 嵌入式 Linux 系统及其应用前景 …… 180

小结…… 181

第 6 章 嵌入式 Linux 开发基础…… 182

6.1 Linux 系统的结构 …… 183

6.1.1 Linux 内核 …… 183

6.1.2 Linux Shell …… 184

6.1.3 Linux 文件系统 …… 185

6.1.4 Linux 实用工具 …… 186

6.2 Linux 常用命令 …… 187

6.2.1 磁盘管理命令…… 187

6.2.2 文件操作命令 …… 189
6.2.3 联机帮助命令 …… 194
6.3 Linux C 编辑、编译、调试工具 …… 195
6.3.1 Linux 下 C 语言编程概述 …… 195
6.3.2 vi 编辑器 …… 197
6.3.3 emacs 编辑器 …… 199
6.3.4 gcc 编译工具 …… 203
6.3.5 gdb 调试工具 …… 205
6.3.6 make 的使用和 Makefile 文件的编写 …… 210
6.3.7 版本控制 …… 214
6.4 Linux C 编程基础 …… 215
6.4.1 Linux 的进程 …… 215
6.4.2 Linux 下的进程控制 …… 216
6.4.3 多线程编程入门 …… 219
6.5 调试程序 …… 222
6.5.1 调试多线程程序 …… 222
6.5.2 调试多进程程序 …… 225
6.6 交叉编译 …… 227
6.6.1 嵌入式系统开发模型 …… 228
6.6.2 交叉编译工具链 …… 228
6.6.3 交叉编译实例 …… 229
小结 …… 230

第 7 章 构建嵌入式 Linux 系统 …… 232
7.1 嵌入式 Linux 系统的构建流程 …… 233
7.1.1 嵌入式 Linux 系统的组成 …… 233
7.1.2 嵌入式 Linux 系统的构建 …… 234
7.2 宿主机和目标机之间的通信 …… 235
7.2.1 宿主机和目标机 …… 235
7.2.2 Windows 的超级终端 …… 235
7.2.3 Linux 的 minicom …… 236
7.2.4 TFTP 协议 …… 238
7.2.5 NFS 网络共享 …… 239
7.3 Bootloader …… 240
7.3.1 Bootloader 的作用 …… 240
7.3.2 Bootloader 的启动方式 …… 241
7.3.3 Bootloader 的两个阶段 …… 241
7.3.4 常用 Bootloader 简介 …… 242
7.4 Linux 内核配置和移植 …… 243
7.4.1 Linux 内核移植准备 …… 243

7.4.2 内核的配置 …… 253
7.4.3 Linux 内核的编译 …… 255
7.5 构建嵌入式根文件系统 …… 256
7.5.1 Linux 下的文件系统 …… 256
7.5.2 嵌入式 Linux 的文件系统 …… 257
7.5.3 Linux 根文件系统目录结构 …… 259
7.5.4 制作根文件系统 …… 262
小结 …… 264

第 8 章 基于 Web 的远程监控系统的设计实例 …… 265
8.1 基于 Web 的远程监控系统简介 …… 266
8.1.1 嵌入式 Web 服务器和远程监控系统 …… 266
8.1.2 基于嵌入式 Web 的远程监控系统应用 …… 267
8.2 系统架构设计 …… 270
8.2.1 网络架构 …… 270
8.2.2 硬件架构设计 …… 271
8.2.3 软件架构设计 …… 272
8.3 系统软件实现 …… 272
8.3.1 嵌入式 Web 服务器的移植和配置 …… 273
8.3.2 HTML 中表单的使用 …… 277
8.3.3 CGI 程序设计 …… 289
8.4 Linux 设备驱动程序设计 …… 302
8.4.1 Linux 下的驱动程序设计基础 …… 302
8.4.2 基于 Linux 2.6 内核的设备驱动程序举例 …… 304
8.5 基于 Web 的 LED 远程控制系统设计 …… 308
8.5.1 LED 驱动程序设计 …… 308
8.5.2 表单设计 …… 311
8.5.3 CGI 程序的编写 …… 312
小结 …… 313

参考文献 …… 315

第1章

嵌入式系统概述

学习目标

通过本章的学习，应该掌握：

- 嵌入式系统的定义、组成、发展概况、应用领域及发展趋势
- 嵌入式系统的开发流程、特点和调试方法
- 常用的嵌入式微处理器和嵌入式操作系统

1.1 嵌入式系统基础

问题： 什么是嵌入式系统？嵌入式系统的发展阶段有哪些？和通用计算机相比嵌入式系统有哪些特点？嵌入式系统由哪些部分组成？嵌入式系统主要应用在哪些领域？
重点： 嵌入式系统的定义，嵌入式系统的组成。
内容： 讲述了嵌入式系统的发展历史、特点、应用领域和发展趋势。

1.1.1 嵌入式系统的发展历史

任务： 认识嵌入式系统的起源，掌握嵌入式系统不同发展阶段的特点。

嵌入式系统起源于微型机时代，近些年来随着网络技术、无线通信、多媒体以及工业自动控制技术的发展，嵌入式技术的发展与应用也日新月异，逐渐成为继个人计算机(Personal Computer，PC)和 Internet 之后信息技术领域的热点。

从 20 世纪 70 年代起，微型机以小型、廉价、低功耗、使用方便、性价比高等特点，得到了广泛的应用，其所具备的智能化功能在工业控制领域内发挥了很大作用，例如，将微型计算机经电气加固与机械加固，另外配置各种外围接口电路，安装到舰船或飞行器构成驾驶辅助系统或是发动机状态检测系统。基于这种需求，传统的计算机便失去了原来的形态和通用的计算机功能。区别于原有的通用计算机系统，将这类为了“专用”的目的而嵌入到对象体系中以实现对象体系智能化控制的计算机系统称为嵌入式计算机系统。

尽管嵌入式系统起源于微型机，由于应用的场合及特点不同，通用计算机与嵌入式计算机之后开始沿两个不同的方向发展。通用计算机的技术要求是高速、海量的数值计算，技术发展方向是总线速度无限提升，存储容量无限扩大。而嵌入式计算机系统的技术要求则是对象的智能化控制能力，技术发展方向是与对象系统密切相关的嵌入性能、控制能力与控制的可靠性。通用计算机系统在体积、价位、可靠性上都无法满足广大对象系统的嵌入式应用要求，因此嵌入式技术的发展必须走独立发展的道路，至今主要经历了无操作系统的单片机(Single Chip Microcomputer，SCM)阶段，以微控制器(Micro Controller Unit，MCU)为基础、以简单操作系统为核心的嵌入式系统阶段，以通用的嵌入式操作系统和系统级芯片(System On a Chip，SOC)为标志的嵌入式系统阶段，面向 Internet 的应用 4 个阶段。

1. 无操作系统的单片机阶段

单片机即单片微型计算机，随着大规模集成电路的出现及发展，计算机的 CPU、RAM、ROM、定时器和多种 I/O 接口集成在一片芯片上，形成芯片级的计算机。1976 年 Intel 公司推出了 MCS-48 单片机，这个只有 1KB ROM 和 64B RAM 的简单芯片成为世界上第一个单片机，同时也开创了将微处理机系统的各种 CPU 外的资源集成到 CPU 芯片上的时代。1980 年 Intel 公司在 MCS-48 的基础上进行全面改进，推出了 8 位的 MCS-51 单片机，获得巨大成功，奠定了嵌入式系统的单片机应用模式，至今 MCS-51 单片机仍在大量使用。

这个阶段是以单片机为核心的控制系统阶段，嵌入式系统大部分应用于一些专业性极强的工业控制系统中，没有操作系统的支持，这一阶段系统的主要特点是：系统结构和功能

都相对单一,处理效率较低,存储容量小,几乎没有用户接口。也是从这个阶段起,嵌入式技术与通用计算机技术开始沿两个不同的方向发展。

2. 以微控制器为基础、以简单操作系统为核心的嵌入式系统阶段

20 世纪 80 年代,随着微电子工艺水平的提高,IC 制造商开始把嵌入式应用中所需要的微处理器、I/O 接口、串行接口以及 RAM、ROM 等部件全部集成到一片超大规模集成电路(Very Large Scale Integration,VLSI)中,制造出面向 I/O 设计的微控制器,并一举成为嵌入式系统领域中异军突起的新秀。与此同时,嵌入式系统的程序员也开始基于一些简单的"操作系统"开发嵌入式应用软件,大大缩短了开发周期,提高了开发效率。

这一阶段嵌入式系统的主要特点是:出现了大量高可靠、低功耗的嵌入式 CPU(如 Power PC 等),各种简单的嵌入式操作系统开始出现并得到迅速发展。此时的嵌入式操作系统虽然还比较简单,但已经初步具有一定的兼容性和扩展性,内核精巧且效率高,主要用来控制系统负载以及监控应用程序的运行。

3. 以通用的嵌入式操作系统和系统级芯片为标志的嵌入式系统阶段

20 世纪 90 年代,随着设计与制作技术的发展,集成电路设计从晶体管的集成发展到逻辑门的集成,现在又发展到 IP 集成,即 SoC 设计技术。SoC 可以有效地降低电子/信息系统产品的开发成本,缩短开发周期,提高产品的竞争力,是未来工业界将采用的最主要的产品开发方式。SoC 通常有以下特征。

(1) 实现复杂系统功能的 VLSI。

(2) 采用超深亚微米工艺技术。

(3) 使用一个以上的嵌入式 CPU/数字信号处理器(Digital Signal Processor,DSP)。

(4) 外部可以对芯片进行编程。

(5) 其主要采用第三方 IP 进行设计。

从 SoC 的特征可以看出,SoC 中包含了微处理器/微控制器、存储器以及其他专业功能逻辑,但并不是包含微处理器、存储器以及其他专业功能逻辑的芯片就是 SoC,如 8051 就集成了微处理器、存储器、时钟部件,但不属于 SoC。SoC 技术被广泛应用的根本原因并不是 SoC 可以集成多少个晶体管,而在于其可以用较短的时间设计出来,这是 SoC 的主要价值所在。

这一阶段系统的主要特点是:嵌入式操作系统能运行在各种不同类型的微处理器上,兼容性好;操作系统内核精小、效率高,并且具有高度的模块化和扩展性;具备文件和目录管理、设备支持、多任务、网络支持、图形窗口以及用户界面等功能;具有大量的应用程序编程接口(Application Programming Interface,API),开发应用程序简单;嵌入式应用软件丰富。

4. 面向 Internet 的应用阶段

面向 Internet 的应用阶段是正在迅速发展的阶段。现在大多数嵌入式系统还孤立于 Internet 之外,但随着 Internet 的发展,Internet 技术和信息家电、工业控制技术结合日益密切,嵌入式设备和 Internet 的结合将是嵌入式系统的未来发展方向。

目前,嵌入式技术与 Internet 技术的结合正在推动着嵌入式技术的飞速发展,嵌入式系统的研究和应用产生了如下新的显著变化。

(1) 新的微处理器层出不穷,嵌入式操作系统自身的结构被设计得更加便于移植,能够在短时间内支持更多的微处理器。

(2) 嵌入式系统的开发成了一项系统工程,开发厂商不仅要提供嵌入式软、硬件系统本

身,同时还要提供强大的硬件开发工具和软件支持包。

(3) 在通用计算机上使用的新技术、新观念开始逐步移植到嵌入式系统中,如嵌入式数据库、移动代理、实时 CORBA 等,嵌入式软件平台得到进一步完善。

(4) 各类嵌入式 Linux 操作系统发展迅速,由于具有源代码开放、系统内核小、执行效率高、网络结构完整等特点,能很好地满足信息家电等嵌入式系统的需要,目前已经能与 Windows CE、Palm OS 等嵌入式操作系统进行有力竞争。

(5) 网络化、信息化的要求随着 Internet 技术的成熟和带宽的提高而日益突出,以往功能单一的设备(如电话、手机、冰箱、微波炉等)的功能不再单一,结构变得更加复杂,网络互联成为必然趋势。

(6) 精简系统内核,优化关键算法,降低功耗和软、硬件成本。

(7) 提供更加友好的多媒体人机交互界面。

综上所述,嵌入式系统技术日益完善,32 位微处理器在该系统中占据主导地位,嵌入式操作系统已从简单走向成熟,开始与 Internet 结合日益密切。美国著名的未来学家葛洛庞帝在 1999 年访华时曾预言,4~5 年后嵌入式系统将是继 PC 和 Internet 之后最伟大的发明。就目前嵌入式技术发展的现状来看,这个预言已经变成了现实,现在嵌入式系统正处于高速发展阶段。

1.1.2 嵌入式系统的定义与特点

任务: 掌握嵌入式系统的定义,嵌入式系统区别于通用计算机系统的特点。

嵌入式系统是一个相对模糊的概念,IEEE 给出的嵌入式系统的定义是:嵌入式系统是用于控制、监视或者辅助操作机器和设备的装置。

在国内的嵌入式系统领域,比较认同的嵌入式系统的概念是:嵌入式系统是以应用为中心,以计算机技术为基础,并且软、硬件可裁剪,适用于对功能、可靠性、成本、体积、功耗有严格要求的应用系统的专用计算机系统。它一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统以及用户的应用程序 4 个部分组成,用于实现对其他设备的控制、监视或管理等功能。

由其定义可以看出,嵌入式系统实际上是嵌入式计算机系统,只是它是嵌入到了更大的、专用的应用系统中的计算机系统,是实际应用系统的一个部件,因此也有人把嵌入式系统定义为:嵌入到对象体系中的专用计算机系统。

含有嵌入式系统的设备称为嵌入式设备,这在生活中随处可见,如电子仪表、手机、PDA、MP3、数字照相机、机顶盒、各种网络设备等。随着技术的进步,嵌入式设备的性能越来越高,其中构成嵌入式系统的主流趋势是 32 位嵌入式处理器加实时多任务操作系统。由以上嵌入式系统的具体应用可见,嵌入式计算机系统是面向具体应用的,要有针对具体应用的"量体裁衣"的软、硬件,与通用计算机系统相比有如下特点。

(1) 嵌入式系统通常是面向特定应用的,嵌入式 CPU 与通用 CPU 最大不同就是嵌入式 CPU 大多工作在为特定用户群设计的系统中,它通常具有功耗低、体积小、集成度高等特点,能够把通用 CPU 中许多由板卡完成的任务集成在芯片内部,从而有利于嵌入式系统设计趋于小型化,移动能力大大增强,跟网络的耦合也越来越紧密。

(2) 大多数嵌入式系统都有实时性要求,在高端应用中,为满足应用需求、增强可靠性和便于开发,通常要有实时多任务操作系统的支持。

(3) 嵌入式系统是将先进的计算机技术、半导体技术和电子技术与各个行业的具体应用相结合后的产物,这一点就决定了它必然是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。

(4) 功耗、成本和可靠性对嵌入式系统具有重要意义。

(5) 嵌入式系统和具体应用有机地结合在一起,它的升级换代也是和具体产品同步进行的,因此嵌入式系统产品进入市场后具有较长的生命周期。

(6) 嵌入式系统本身不具备自主开发能力,设计完成以后用户通常不能再对其中的程序进行修改,而是必须使用一套开发工具在适当的环境下进行开发。

(7) 从某种意义上来说,通用计算机行业的技术是垄断的。嵌入式系统则不同,嵌入式系统工业是不可垄断的高度分散的工业,充满了竞争、机遇与创新,没有哪一个系列的处理器和操作系统能够垄断全部市场,即便在体系结构上存在着主流,各不相同的应用领域决定了不可能由少数公司、少数产品垄断全部市场,因此嵌入式系统领域的产品和技术必然是高度分散的,留给各个行业高新技术公司的创新余地很大。另外,社会上的各个应用领域是不断向前发展的,要求其中的嵌入式处理器核心也同步发展,这也形成了推动嵌入式工业发展的强大动力。

1.1.3 嵌入式系统的组成

任务: 了解嵌入式系统的组成及各部分的作用。

嵌入式系统是专用计算机应用系统,与一般计算机一样,也是由硬件和软件组成的。硬件部分包括嵌入式微处理器(MPU)或微控制器(MCU)以及外围硬件。软件部分包括嵌入式操作系统和应用程序。另外为了开发嵌入式系统,开发环境也是必不可少的。这几个部分之间的关系如图 1-1 所示。

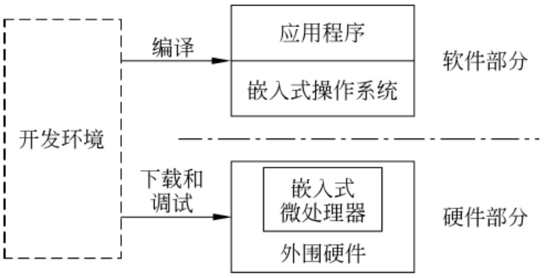


图 1-1 嵌入式系统的组成

嵌入式微处理器就是嵌入到应用对象系统中的专用处理器,相对于通用 CPU(如 x86 系列)而言,一般对价格、尺寸、功耗等方面的限制比较多,目前世界上具有嵌入式功能特点的处理器已经超过 1000 种,流行体系结构包括 MCU、MPU 等 30 多个系列。

外围硬件是嵌入式微处理器以外的硬件,它提供了系统中需要但嵌入式微处理器内部不具有的功能模块,如时钟、电源、内存、各种通信端口等。

嵌入式微处理器和外围硬件构成了系统的硬件基础。

嵌入式操作系统(Embedded Operating System,EOS)负责嵌入式系统的全部软、硬件资源的分配、调度工作,控制协调并发活动。EOS 是相对于一般操作系统而言的,它除了具备一般操作系统最基本的功能,如任务调度、同步机制、中断处理、文件功能等外,还有可裁剪性、强实时性、统一的接口、固化代码、良好的移植性等特点。

应用程序决定了系统具体实现的功能,在嵌入式系统中,由于控制硬件是嵌入式系统的基本操作,因此嵌入式应用程序依然与系统硬件关系密切。尤其在没有操作系统的情况下,嵌入式应用程序需要直接访问寄存器或者设备的地址来操作硬件。

开发环境并不是嵌入式系统产品的一部分,但是它在嵌入式系统产品开发过程中起着至关重要的作用。嵌入式开发环境包括:嵌入式交叉编译环境、主机到目标机的程序载入环境、主机到目标机的调试环境和主机的仿真环境等。

1.1.4 嵌入式系统的应用领域

任务:了解嵌入式系统在不同领域中的应用。

嵌入式系统在现代人们的生活中无处不在,常见的有手机、PDA、智能家电、数码产品(如 DC、DV、MP4)等。嵌入式系统的存在大大丰富了人们的日常应用,并逐渐与移动通信、传感器网络技术等结合起来改变现有的计算环境。未来的嵌入式应用将深入到人们的日常生活、工作学习、工业领域、日常消费品以及军事领域中。

1. 工业控制

基于嵌入式芯片的工业自动化设备将获得长足的发展,目前已经有大量的 8 位、16 位和 32 位嵌入式微控制器正在应用中,如工业过程控制、数字机床、电力系统、电网安全、电网设备监测、石油化工系统。就传统的工业控制产品而言,低端型采用的往往是 8 位单片机,但是随着技术的发展,32 位、64 位的处理器逐渐成为工业控制设备的核心,在未来几年内必将获得长足的发展。

2. 信息家电

信息家电将成为嵌入式系统最大的应用领域,冰箱、空调等家用电器的网络化、智能化将引领人们的生活步入一个崭新的空间。例如水、电、煤气表的远程自动抄表,安全防火、防盗系统,其中嵌入的专用控制芯片将代替传统的人工检查,并实现更高、更准确和更安全的性能。

3. 移动计算设备和网络设备

移动计算设备包括手机、PDA、掌上电脑等,中国拥有数量最大的手机用户,而掌上电脑(或 PDA)由于易于使用、携带方便、价格便宜,未来几年将在我国得到快速发展。PDA 与手机已呈现融合趋势,用掌上电脑(或 PDA)上网,人们可以随时随地获取信息。

网络设备包括路由器、交换机、Web 服务器网络接入盒等,设计和制造嵌入式瘦服务器、嵌入式网关和嵌入式因特网路由器已成为嵌入式 Internet 时代的关键和核心技术。

4. 汽车电子

汽车电子产品可分为两大类:汽车电子控制装置,包括动力总成控制、底盘和车身电子控制、舒适和防盗系统;车载汽车电子装置,包括汽车信息系统(车载电脑)、导航系统、汽车视听娱乐系统、车载通信系统、车载网络等。汽车嵌入式系统近年来发展非常迅速,基于网络通信和实时多任务并行处理的嵌入式高端应用将会越来越广泛。汽车嵌入式系统在硬件上采用 32 位或 64 位高性能处理器,在软件上嵌入实时操作系统,具有功能多样、集成度高、通信网络化、开发快捷及成本低廉的特点,在汽车电子控制和车载网络通信系统方面有着广

泛的应用。

5. 机器人

机器人技术的发展从来都是与嵌入式系统的发展紧密联系在一起的,近年来由于嵌入式处理器的高速发展,机器人从硬件到软件也呈现了新的发展趋势。例如,火星车采用风河公司的 Vxworks 嵌入式操作系统,可以在不与地球联系的情况下自主工作。1997 年美国发射的“索杰纳”火星车带有机械手,可以采集火星上的各种地况,并且通过摄像头把火星上的图像发回地面指挥中心,在火星上自主工作了 3 个月。随着嵌入式控制器越来越微型化、功能化,微型机器人、特种机器人等也将有更大的发展。

6. 军事领域

嵌入式系统在军事领域中的应用最早出现在 20 世纪 60 年代的武器控制系统之中,主要用于各种武器的控制系统(火炮控制、导弹控制、智能炸弹制导引爆装置等),坦克、舰艇、军用轰炸机等各种海陆空军用电子装备,雷达、电子对抗军用通信设备,野战指挥作战用的各种专用设备中。新型武器装备的研制以及现有武器的改造都会涉及嵌入式系统的开发与升级,军用嵌入式系统也正在朝着更加智能化和网络化的方向快速发展。

1.1.5 嵌入式技术的发展趋势

任务:了解当前嵌入式系统技术的发展趋势。

1. 32 位处理器成为市场主流

随着 ARM 处理器在全球范围的流行,32 位的 RISC 嵌入式处理器已经成为嵌入式应用和设计的主流。与国内大量应用的 8 位单片机相比,32 位的嵌入式 CPU 有着更大的优势,它为嵌入式设计带来丰富的硬件功能和额外的性能,使得整个嵌入式系统的升级只需通过软件的升级即可实现。而 8 位处理器通常受到的 64KB 软件限制也不存在了,设计者可以任意选择多任务操作系统,并将应用软件设计得复杂庞大,真正体现“硬件软件化”的设计思想。

2. 小型化、低成本、低功耗

随着嵌入式系统越来越多地应用于移动和无线终端产品,嵌入式系统对产品的小型化、低成本、低功耗需求越来越明显。不同于通用计算机,功耗对于嵌入式系统是一个重要考虑因素,很多嵌入式设备采用电池进行供电。另外大功耗还会影响到电源寿命及散热问题。嵌入式系统的成本是由有硬件成本和软件成本两方面组成的,硬件成本是由嵌入式微处理器及其外围硬件构成的,软件成本则与软件开发过程选择的技术相关。

3. 人性化的人机界面

人机界面(Human Machine Interface,HMI)连接可编程控制器(Programmable Logical Controller,PLC)、变频器、直流调速器、仪表等工业控制设备,利用显示屏显示,通过输入单元(如触摸屏、键盘、鼠标等)写入工作参数或输入操作命令,可以实现人与机器的信息交互。人机界面是近年来嵌入式技术在自动化和控制领域发展与变化的亮点。软、硬件技术的进步大大推动了人机界面显示器与所控制系统的复杂性与精确度的提高。图像和动画功能日益丰富,能够处理的任务更复杂,无线连接功能不断增强,这些都有助于推动人机界面应用的不断发展。

4. 完善的开发平台

嵌入式系统的更新变化越来越快,嵌入式系统设计开发工程师面临着强烈的市场需求

以及日益错综复杂的设计挑战,对开发时间要求比较紧,尤其是消费类产品,更是要求快速开发、生产、上市。正确选择一套先进的、功能强大的,同时又使用方便、界面友好的开发工具平台就显得至关重要。随着嵌入式系统市场对产品开发的需求不断增加,嵌入式开发平台也日臻完善,目前已形成了嵌入式开发的硬件平台,包括开发板、调试器、仿真器、烧写器等硬件及其配套的软件集成开发环境。

5. 可编程逻辑器件的兴起

可编程逻辑器件(FPGA)一直以其设计灵活以及现场可编程的特性,在市场上稳稳固守着一席之地,随着半导体制造工艺的进步,器件集成度越来越高,其应用也日益复杂。过去应用 FPGA 的主要是硬件设计人员,他们对器件本身的物理结构及特性都有相当的了解,而如今系统集成工程师、DSP 开发人员甚至嵌入式软件工程师也都需要在可编程逻辑器件平台上进行系统开发,FPGA 的复杂性对他们将是一大挑战。

6. 产品提供强大的网络服务功能

目前已有许多嵌入式系统将内置网络功能视为系统基本特性的发展趋势,根据 Forrester Research 的研究显示,目前 95%的网络接入设备不是通用计算机,而是带有网络功能的嵌入式系统,也就是具备无线机器对机器(M2M)功能的嵌入式网络解决方案。由于越来越多的设备需要通过 Internet 进行通信或者控制,因此互联和安全功能成为除操作系统之外,设计者们需要考虑的主要设计需求。嵌入式系统的工程师们在设计的前期就要仔细选择互联和安全协议栈,确保 TCP/IP 和相应的安全与移动特性能够在该嵌入式系统的生命周期内,满足大部分功能升级和设计更新的需要。

1.2 嵌入式系统的硬件组成

问题: 什么是嵌入式微处理器?常见的嵌入式外围设备接口有哪些?列举常用的嵌入式微处理器及其应用场景。

重点: 嵌入式微处理器的定义,嵌入式外围接口。

内容: 讲述了嵌入式系统的硬件组成,包括嵌入式处理器、嵌入式外围设备与接口、典型的嵌入式处理器与开发板。

嵌入式系统的硬件是嵌入式系统软件运行的基础,它提供了嵌入式系统软件运行的物理平台和通信接口。嵌入式系统的硬件组成如图 1-2 所示。

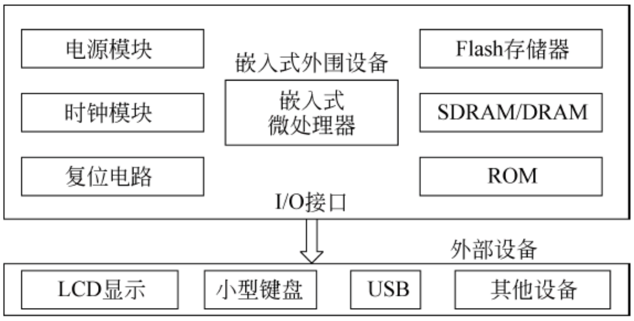


图 1-2 嵌入式系统的硬件组成

1.2.1　嵌入式处理器

任务：理解嵌入式处理器的特点，了解嵌入式处理器的分类。

嵌入式系统的核心是嵌入式微处理器。嵌入式微处理器一般具备以下 4 个特点。

（1）对实时多任务有很强的支持能力，能实时完成多任务并且有较短的中断响应时间，从而使内部的代码和实时内核的执行时间减少到最低限度。

（2）具有功能很强的存储区保护功能。这是由于嵌入式系统的软件结构已模块化，为了避免在软件模块之间出现错误的交叉作用，需要设计强大的存储区保护功能，同时也有利于诊断软件问题。

（3）具有可扩展的处理器结构，以便快速地扩展出满足应用的高性能嵌入式微处理器体系。

（4）嵌入式微处理器必须功耗很低，尤其是用在便携式的无线及移动的计算和通信设备中靠电池供电的嵌入式系统更是如此，如需要的功耗只有 mW 级甚至 μW 级。

目前，嵌入式处理器可以分成下面几类。

1. 嵌入式微处理器

嵌入式微处理器是由通用计算机中的 CPU 演变而来的。它的特征是具有 32 位以上的处理器，具有较高的性能，当然其价格也相应较高。与计算机处理器不同的是，在实际嵌入式应用中，只保留和嵌入式应用紧密相关的功能硬件，去除其他的冗余功能部分，这样就以最低的功耗和资源实现嵌入式应用的特殊要求。和工业控制计算机相比，嵌入式微处理器具有体积小、重量轻、成本低、可靠性高的优点。目前主要的嵌入式处理器类型有 Am186/88、386EX、SC-400、Power PC、68000、MIPS、ARM/StrongARM 系列等。其中 ARM/StrongARM 是专为手持设备开发的嵌入式微处理器，价位属于中档。

2. 嵌入式微控制器

嵌入式微控制器的典型代表是单片机，从 20 世纪 70 年代末单片机出现到今天，虽然已经经过了 30 多年的历史，但这种 8 位的电子器件目前在嵌入式设备中仍然有着极其广泛的应用。单片机芯片内部集成 ROM/EPROM、RAM、总线、总线逻辑、定时/计数器、看门狗、I/O、串行口、脉宽调制输出、A/D、D/A、Flash RAM、EEPROM 等各种必要功能和外设。和嵌入式微处理器相比，微控制器的最大特点是单片化，体积小，从而使功耗和成本下降，可靠性提高。微控制器是目前嵌入式系统工业的主流。微控制器的片上外设资源一般比较丰富，适合于进行控制，因此称为微控制器。

由于 MCU 价格低廉，功能优良，所以其拥有的品种和数量最多，比较有代表性的包括 8051、MCS-251、MCS-96/196/296、P51XA、C166/167、68K 系列以及 MCU 8XC930/931、C540、C541，并且有支持 I^2C、CAN-Bus、LCD 的 MCU 及众多专用的 MCU 和兼容系列。目前 MCU 占嵌入式系统约 70%的市场份额。近来 Atmel 公司出产的 AVR 单片机由于集成了 FPGA 等器件，具有很高的性价比，势必将推动单片机获得更快的发展。

3. 嵌入式 DSP 处理器

嵌入式 DSP 处理器（Embedded Digital Signal Processor，EDSP）专门用于信号处理，在系统结构和指令算法方面进行了特殊设计，具有很高的编译效率和指令执行速度。在数字滤波、FFT、谱分析等各种仪器上 DSP 得到了大规模的应用。

DSP 的理论算法在 20 世纪 70 年代就已经出现，但是由于专门的 DSP 处理器还未出现，所以这种理论算法只能通过 MPU 等由分立元件实现。MPU 较低的处理速度无法满足 DSP 的算法要求，其应用领域仅仅局限于一些尖端的高科技领域。随着大规模集成电路技术的发展，1982 年世界上诞生了首枚 DSP 芯片。其运算速度比 MPU 快了几十倍，在语音合成和编码解码器中得到了广泛应用。至 20 世纪 80 年代中期，随着 CMOS 技术的进步与发展，第二代基于 CMOS 工艺的 DSP 芯片应运而生，其存储容量和运算速度都得到成倍提高，成为语音处理、图像硬件处理技术的基础。到 20 世纪 80 年代后期，DSP 的运算速度进一步提高，应用领域也从上述范围扩大到了通信和计算机方面。20 世纪 90 年代后，DSP 发展到了第 5 代产品，集成度更高，使用范围也更加广阔。

目前应用最为广泛的是 TI 的 TMS320C2000/C5000 系列，另外如 Intel 的 MCS-296 和 Siemens 的 TriCore 也有各自的应用范围。

4. 嵌入式片上系统

SoC 是追求产品系统最大包容的集成器件，是目前嵌入式应用领域的热门话题之一。SoC 最大的特点是成功实现了软、硬件无缝结合，直接在处理器片内嵌入操作系统的代码模块，而且 SoC 具有极高的综合性，可以在一个硅片内部运用 VHDL 等硬件描述语言，实现一个复杂的系统。用户不需要再像传统的系统设计一样，绘制庞大复杂的电路板，一点点连接焊制，只需要使用精确的语言，综合时序设计直接在器件库中调用各种通用处理器的标准，进行仿真之后就可以直接交付芯片厂商进行生产。由于绝大部分系统构件都放置在系统内部，整个系统特别简洁，不仅减小了系统的体积和功耗，而且提高了系统的可靠性，提高了设计生产效率。

由于 SoC 往往是专用的，所以大部分都不为用户所知，比较典型的 SoC 产品是 Philips 的 Smart XA。少数通用系列如 Siemens 的 TriCore，Motorola 的 M-Core，某些 ARM 系列器件，Echelon 和 Motorola 联合研制的 Neuron 芯片等。一些大的芯片公司将通过推出成熟的、能占领多数市场的 SoC 芯片，在声音、图像、影视、网络及系统逻辑等应用领域中发挥重要作用。

1.2.2 嵌入式外围设备与接口

任务： 了解嵌入式系统中常见的嵌入式外围设备，嵌入式接口的作用及常见接口。

1. 嵌入式外围设备

嵌入式外围设备指在嵌入式系统中用于完成存储、通信、调试、显示等辅助功能的部件，常见的嵌入式外围设备有以下几种。

（1）实时时钟

提供可靠的时钟信息，包括时分秒和年月日，即使系统处于关机或停电状态，实时时钟通过后备电池供电也能继续正常工作。一些嵌入式处理器（如 S3C2410 处理器）芯片内部集成了实时时钟单元；未集成时，则需要外扩实时时钟芯片，典型的只需要一个高精度的晶体振荡器（简称晶振）。

（2）存储设备

存储设备提供执行程序和存储数据所需的空间，常见的有 RAM（Random Access

Memory)、ROM(Read-Only Memory)和闪速存储器(Flash Memory)。其中 RAM 和 ROM 用来提供程序执行的空间,通常采用半导体器件,具有密度大、体积小、访问速度快、性能可靠、使用寿命长等优点。闪速存储器用来存储程序和数据,掉电后数据不会丢失。闪速存储器是一种非易失性存储器(Non-Volatile Memory,NVM),根据结构的不同可以将其分成 NOR Flash和 NAND Flash 两种。

(3) 输入设备

在嵌入式系统中,输入设备向计算机输入数据和信息,是计算机与用户或其他设备通信的桥梁。常用的如收款机系统中的矩阵式小型键盘,由几个简单的数字键和功能键组成。触摸屏也常用于手机等移动通信终端,在液晶屏上叠加一片触摸屏,用触控笔或手指头直接点选按键或输入文字,具有轻薄短小、便于携带、使用方便的优点。

(4) 输出设备

输出设备用于输出数据,是人与计算机交互的一种部件,可以把各种数据或信息以数字、字符、图像、声音等形式表示出来。嵌入式系统中常见的输出设备有 LED 和 LCD。LED 作为电源指示灯、电平指示器、工作状态显示器或微光源,形式上有数码管、符号管、米字管、点阵显示屏,耗电少,成本低,配置简单灵活,安装方便,耐振动,寿命长。LCD 利用液晶同时具备的固态晶体的光学特性和液态物质的流动特性,与 CRT 相比具有体积小、重量轻、辐射小的优势。

2. 嵌入式系统接口

嵌入式系统接口用于连接和扩展系统部件,包括并行接口与串行接口。并行接口是指数据的各位同时进行传送,其特点是传输速率高,但当传输距离较远、位数又多时,将导致通信线路很复杂且成本提高,传输总线的长度受限(过长时,电子线路间将产生电容效应),且抗干扰能力差,如打印机并口(Parallel Port)。串行接口简称串口,也称为串行通信接口(通常指 COM 接口),是采用串行通信方式的扩展接口。一条信息的各位数据被逐位按顺序传送的通信方式称为串行通信。串行通信的特点是:数据位逐位按顺序传送,最少只需一根传输线即可完成,成本低但传送速度慢。串行通信的距离可以从几米到几千米。根据信息的传送方向,串行通信可以进一步分为单工、半双工和全双工 3 种。在嵌入式系统中常见的串行接口包括 I^2C、I^2S、USB、IEEE 1394 等。

1.2.3 典型的嵌入式处理器与开发板

任务:了解典型的嵌入式处理器的种类及特点。

嵌入式处理器是嵌入式系统的核心,是控制、辅助系统运行的硬件单元。范围极广,从最初的 4 位处理器、目前仍在大规模应用的 8 位单片机,到最新的受到广泛青睐的 32 位、64 位嵌入式 CPU。

1. ARM

ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司,世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核,根据各自不同的应用领域,加入适当的外围电路,形成自己的 ARM 微处理器芯片,从而进入市场。ARM 处理器本身是 32 位设计,但也配备 16 位指令集,一般来讲存储器比等价 32 位代码节省达 35%,却保留了 32 位系统的所有优

势。ARM 的 Jazelle 技术使得 Java 加速核获得比基于软件的 Java 虚拟机(Java Virtual Machine,JVM)高得多的性能,和同等的非 Java 加速核相比功耗降低 80%。另外,ARM 在提供通用的 RISC 处理器架构的同时,还为其增添了一些针对特定应用的高性能指令集,比如,ARM 为信号处理算法专门发布了 V5TE 的架构,在普通的 ARM 架构基础上扩展了数字信号处理(DSP)指令集,使得 ARM 的 CPU 系列能够更好的适应复杂的信号处理。

2. PowerPC

PowerPC 是由 IBM、Motorola 和 Apple 联合开发的高性能 32 位和 64 位 RISC 微处理器系列,以与垄断 PC 市场的 Intel 微处理器相竞争。PowerPC 处理器是 RISC 嵌入式应用的理想基础平台,PowerPC 微处理器自从 1994 年推出,凭借其出色的性能、高度整合和技术先进的特性在网络通信、工业控制、家用数字化、网络存储、军工和电力系统控制等领域都具有非常广泛的应用。

3. MIPS

MIPS 公司是一家设计制造高性能、高档次及嵌入式 32 位和 64 位处理器的厂商,在 RISC 处理器方面占有重要地位。MIPS 公司设计 RISC 处理器始于 20 世纪 80 年代初,1986 年推出 R2000 处理器,1988 年推出 R3000 处理器,1991 年推出第一款 64 位商用微处理器 R4000。之后又陆续推出 R8000(于 1994 年)、R10000(于 1996 年)和 R12000(于 1997 年)等型号。随后,MIPS 公司的战略发生变化,把重点放在嵌入式系统上。1999 年,MIPS 公司发布 MIPS32 和 MIPS64 架构标准,为未来 MIPS 处理器的开发奠定了基础。新的架构集成了所有原来的 MIPS 指令集,并且增加了许多更强大的功能。MIPS 公司陆续开发了高性能、低功耗的 32 位处理器内核(Core)MIPS32 4Kc 与高性能的 64 位处理器内核 MIPS64 5Kc。2000 年,MIPS 公司发布了针对 MIPS32 4Kc 的版本以及 64 位 MIPS64 20Kc 处理器内核。

4. Intel Atom

Intel Atom(中文:凌动,开发代号:Silverthorne)是 Intel 的一个处理器系列。处理器采用 45nm 工艺制造,集成 4700 万个晶体管。L2 缓存为 512KB,支持 SSE3 指令集和 VT 虚拟化技术(部分型号)。与一般的桌面处理器不同,Atom 处理器采用顺序执行结构,这样做可以减少晶体管的数量。Atom 处理器系列有 6 个型号,全部属于 Z500 系列。它们分别是 Z500、Z510、Z520、Z530、Z540 和 Z550。最低端的 Z500 内核频率是 800MHz,FSB 则是 400MHz。而最高速的 Z550,内核频率是 2.0GHz,FSB 则是 533MHz。从 Z520 开始,所有的处理器都支持超线程技术,但只增加了不到 10%的耗电。

5. AVR 系列单片机

AVR 单片机是 Atmel 公司于 1997 年推出的 RISC 单片机,AVR 系列单片机都具备 1MIPS/MHz(每秒百万条指令/兆赫兹)的高速处理能力。AVR 单片机吸收了 DSP 双总线的特点,采用 Harvard 总线结构,因此单片机的程序存储器和数据存储器是分离的,并且可对具有相同地址的程序存储器和数据存储器进行独立的寻址。AVR 单片机采用低功率、非挥发的 CMOS 工艺制造,除具有低功耗、高密度的特点外,还支持低电压的联机 Flash、EEPROM 写入功能,支持 BASIC、C 等高级语言编程,具有多个系列,包括 ATtiny、AT90、ATmega,每个系列又包括多个产品,它们在功能和存储器容量等方面有很大的不同,但基本结构和原理都类似,而且编程方法也相同。

6. MCS-51 系列单片机

MCS-51 单片机是美国 Intel 公司于 1980 年推出的产品，与 MCS-48 单片机相比，它的结构更先进，功能更强，在原来的基础上增加了更多的电路单元和指令，指令数达 111 条，MCS-51 单片机可以算是相当成功的产品，一直到现在，MCS-51 系列或与其兼容的单片机仍是应用的主流产品。MCS-51 系列单片机主要包括 8031、8051 和 8751 等通用产品，以其典型的结构和完善的总线专用寄存器的集中管理，众多的逻辑位操作功能及面向控制的丰富的指令系统，为之后其他单片机的发展奠定了基础。正因为其优越的性能和完善的结构，后来的许多厂商多沿用或参考了其体系结构，有许多大的电气厂商丰富和发展了 MCS-51 单片机，如 Philips、Dallas、Atmel 等著名的半导体公司都推出了兼容 MCS-51 的单片机产品，台湾 Winbond 公司也推出了兼容的 C51，习惯上将 MCS-51 简称为 C51。

7. MC68000

Motorola68000 型中央处理器，又称为 MC68000，是由美国 Motorola 公司(其半导体部门现已独立成为飞思卡尔公司)出品的一款 16/32 位 CISC(复杂指令集)微处理器。作为 M68K 处理器系列的第一个成员，MC68000 于 1979 年投放市场。由于内部使用 32 位总线和寄存器，它在软件层(指令集)与随后的纯 32 位产品基本上保持兼容。目前这款微处理器在嵌入式领域仍在应用。

1.3 嵌入式系统的软件组成

问题： 和通用计算机相比，嵌入式软件的基本特点有哪些？嵌入式软件开发为什么使用交叉开发模式？市场上常见的嵌入式操作系统有哪些？
重点： 交叉开发环境，嵌入式操作系统，嵌入式软件开发的特点。
内容： 讲述了嵌入式系统的软件组成，包括嵌入式软件的基本特点和分类、嵌入式系统软件的组成、开发环境、开发要点，并简单介绍了嵌入式操作系统的概念、特点及作用。

1.3.1 嵌入式软件的基本特点与分类

任务： 掌握嵌入式软件的特点，嵌入式软件的层次划分及作用。

嵌入式应用软件是实现嵌入式系统功能的关键，对嵌入式处理器系统软件和应用软件的要求也和通用计算机有所不同。嵌入式软件主要有以下特点。

(1) 软件要求固态化存储。为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在存储器芯片或单片机本身中，而不是存储在磁盘等载体中。

(2) 软件代码的质量和可靠性高。尽管半导体技术的发展使处理器速度不断提高，片上存储器容量不断增加，但在大多数应用中，存储空间仍然是宝贵的，还存在实时性的要求，为此对编写的程序和编译工具的质量要求高，以减少程序二进制代码的长度，提高执行速度。

(3) 许多应用要求系统软件具有实时处理能力。在多任务嵌入式系统中，对重要性各不相同的任务进行统筹兼顾的合理调度是保证每个任务及时执行的关键。这种任务调度单纯通过提高处理器速度是无法完成和没有效率的，只能由嵌入式操作系统来完成，因此要求

操作系统具有实时处理能力。

(4) 多任务操作系统是知识集成的平台和走工业标准化道路的基础。

(5) 嵌入式系统软件采用 C 语言开发将是最佳选择。

嵌入式系统软件主要包含以下几个部分。

(1) 嵌入式操作系统。嵌入式操作系统(Embedded Operating System)是一种实时的、支持嵌入式系统应用的操作系统软件,它是嵌入式系统(包括硬、软件系统)的重要组成部分,通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面、标准化浏览器等。

嵌入式操作系统具有通用操作系统的基本特点,如能够有效管理越来越复杂的系统资源;能够把硬件虚拟化,使得开发人员从繁忙的驱动程序移植和维护中解脱出来;能够提供库函数、标准设备驱动程序以及工具集等。与通用操作系统相比较,嵌入式操作系统在系统实时高效性、硬件的相关依赖性、软件固态化以及应用的专用性等方面具有较为突出的特点。

(2) 嵌入式应用软件。嵌入式应用软件是针对特定应用领域,基于某一固定的硬件平台,用来达到用户预期目标的计算机软件。

由于用户任务可能有时间和精度上的要求,因此有些嵌入式应用软件需要特定嵌入式操作系统的支持。嵌入式应用软件和普通应用软件有一定的区别,它不仅要求准确性、安全性和稳定性等能够满足实际应用的需要,而且还要尽可能地进行优化,以减少对系统资源的消耗,降低硬件成本。目前市场上已经出现了各式各样的嵌入式应用软件,包括浏览器、E-mail 软件、文字处理软件、通信软件、多媒体软件、个人信息处理软件、智能人机交互软件、各种行业应用软件等。

(3) 硬件抽象层。硬件抽象层(Hardware Abstraction Layer,HAL)是位于操作系统内核与硬件电路之间的接口层,用于将硬件抽象化。

也就是说,可通过程序来控制所有硬件电路如 CPU、I/O、存储器等的操作,这样就使得系统的设备驱动程序与硬件设备无关,从而大大地提高了系统的可移植性。从软、硬件测试的角度来看,软、硬件的测试工作都可分别基于硬件抽象层来完成,使得软、硬件测试工作的并行进行成为可能。在定义抽象层时,需要规定统一的软、硬件接口标准,其设计工作需要基于系统需求来做,代码工作可由对硬件比较熟悉的人员来完成。抽象层一般应包含完成相关硬件的初始化、数据的输入/输出操作、硬件设备的配置操作等功能。

(4) 板级支持包。板级支持包(Board Support Package,BSP)在嵌入行业中用来代表在一个特殊硬件平台上快速构建一个嵌入操作系统所需的原始资料或者二进制软件包。

BSP 的作用是支持操作系统,使之能够更好地运行在硬件平台上。BSP 是相对于操作系统而言的,不同的操作系统对应于不同定义形式的 BSP,包括 Windows CE、Linux、VxWorks 等。SoC/CPU 厂商应向其芯片的用户提供一个基本的 BSP 包,以支持主板厂商或整机制造厂商在此基础上定制和开发各种商用终端产品。

(5) 设备驱动程序。设备驱动程序(device driver),简称驱动程序(driver),是一个允许高级(high level)计算机软件(computer software)与硬件(hardware)交互的程序。

这种程序建立了一个硬件与硬件,或硬件与软件交互的界面,经由主板上的总线(bus)或其他沟通子系统(subsystem)与硬件形成连接的机制,这样的机制使得硬件设备(device)上的数据交换成为可能。不同于在通用计算机上的开发,进行嵌入式系统开发时通常需要

对每一个外围设备进行开发或移植驱动程序。

(6) 操作系统的应用程序接口函数。

操作系统的应用程序接口函数(Application Programming Interface,API)其实就是操作系统留给应用程序的一个调用接口,应用程序通过调用操作系统的 API 而使操作系统去执行应用程序的命令(动作)。

1.3.2 嵌入式软件开发环境

任务: 了解常见的嵌入式软件集成开发环境及工具。

嵌入式系统通常为一个资源受限的系统,直接在嵌入式系统的硬件平台上编写软件比较困难,有时甚至是不可能的。一般采用的办法是,先在通用计算机上编写程序;然后通过交叉编译,生成目标平台上可运行的二进制代码格式;最后下载到目标平台上的特定位置运行。常见的软件开发工具如下。

(1) GNU 工具链

目前已经能够支持 x86、ARM、MIPS、PowerPC 等多种处理器,GNU 工具链(GNU tool chain)是一个包含了由 GNU 项目所产生的各种编程工具的集合。这些工具形成了一条工具链(串行使用的一组工具),用于开发应用程序和操作系统。GNU 工具链在针对嵌入式系统的 Linux 内核、BSD 及其他软件的开发中起着至关重要的作用。GNU 工具链中的部分工具也可被 Solaris、Mac OS X、Microsoft Windows(通过 Cygwin 与 MinGW/MSYS)、Sony PlayStation 3 等其他平台直接使用或进行移植。GNU 工具链中包含如下项目。

GNU make:用于编译和构建的自动工具。

GNU 编译器集合(GNU Compiler Collection,GCC):一组编程语言的编译器。

GNU Binutils:包含链接器、汇编器和其他工具的工具集。

GNU Debugger(GDB):代码调试工具。

GNU 构建系统(autotools):Autoconf、Autoheader、Automake、Libtool。

(2) ARM Developer Suite

ARM Developer Suite™(ADS)是全套的实时开发软件工具包编译器,生成的代码密度和执行速度优异,可快速创建基于 ARM 体系结构的应用。ADS 包括 3 种调试器:ARM eXtended Debugger(AXD)、向下兼容的 ARM Debugger for Windows/ARM Debugger for UNIX 和 ARM 符号调试器,其中 AXD 不仅拥有低版本 ARM 调试器的所有功能,还新添了图形用户界面、更方便的视窗管理、数据显示格式化和编辑以及全套的命令行界面,该产品还包括 RealMonitor。ARM 的 Real-Time Trace™和 RealMonitor 均为重要的实时调试解决方案,能够缩短开发周期,提供特殊软件调试功能,可运行于带深度嵌入处理器内核的高集成系统芯片中。Real-Time Trace 产品包括跟踪调试工具 MultiTrace、嵌入式跟踪宏单元和 Multi-ICE。RealMonitor 包括 RMTarget™。RMHost™ 是 ARM Developer Suite 的补充硬件。

(3) WindRiver Tornado

WindRiver 推出了 Tornado Ⅱ型开发平台,显著地提高了嵌入式开发商的产品化时间,

该指标是实现快速上市的关键因素。Tornado Ⅱ型的集成元件包括 Tornado 工具、VxWorks 运行系统以及一整套将目标连接至主机的通信软件。Tornado 工具为内容广泛的核心及软件交叉开发工具以及实用程序套件，而 VxWorks 运行系统是一种高性能、可扩展的实时操作系统，在目标处理器上运行。

（4）Microsoft Embedded Visual C++

Embedded Visual C++（EVC）是微软公司提供的开发嵌入式软件的平台，是 Visual C++ 的子集，EVC 自带了标准的 Windows CE 的 SDK，可以从 Platform Builder 中导出 SDK，然后安装在 EVC 中。EVC 和 Platform Builder 的不同之处在于：Platform Builder 针对的是操作系统的定制，编译目标是整个 OS 的内核，虽然 Platform Builder 也能开发应用程序，但是这些应用程序是作为整个系统的一部分而存在的，编译时还是以一个操作系统为单位；EVC 针对的是应用软件的开发，特定操作系统的 SDK 为它提供了系统的运行环境，编译时以一个应用程序为单位，EVC 会提供虚拟器加载 SDK 来运行应用程序。

1.3.3 嵌入式软件开发的要点

任务： 嵌入式软件开发为软、硬件综合开发，掌握注意要点及区别于通用计算机软件开发需要考虑的问题。

嵌入式系统包含硬件和软件两部分，嵌入式系统开发的最大特点就是需要软、硬件综合开发，软件又是嵌入式系统的核心。在嵌入式系统中，软件和硬件紧密配合，协调工作，共同完成系统预定的功能。与通用机相比，嵌入式系统属于专用系统，资源有限，需要借助相应的开发工具才能有效地进行开发。在嵌入式系统的开发中，需要考虑的主要因素有软、硬件协同设计、嵌入式处理器的选择、操作系统的选择、嵌入式系统的交叉开发环境和调试等。

1. 软、硬件协同设计

传统的嵌入式开发，硬件和软件分为两个独立的部分，只能分别改善硬件、软件的性能，不可能达到整个系统的最优。在嵌入式系统中，软件和硬件是紧密配合、协调工作的，从理论上来说，对于每一个应用系统，都存在一个适合该系统的硬件、软件最佳组合，这就产生了一种全新的发展中的设计理论：软、硬件协同设计。这种方法对软、硬件进行统一表示和功能划分，设计时考虑软、硬件的实现，以取得最好的效果。软、硬件协同设计的过程可归纳为如下几个步骤。

（1）需求分析。

（2）软、硬件协同设计。

（3）软、硬件实现。

（4）软、硬件协同测试和验证。

这种方法的特点是协同设计（co-design）、协同测试（co-test）和协同验证（co-verification）。它充分考虑了软、硬件的关系，并在设计的每个层面上给予测试验证，可以尽早发现和解决问题，避免灾难性的错误出现，提高开发效率。

2. 嵌入式处理器的选择

在嵌入式系统的硬件中，嵌入式处理器是整个系统的核心部件，其性能的好坏直接决定整个系统的运行效果。由于应用各不相同，嵌入式处理器种类多，因此选择一款合适的处理

器并不是一件容易的事，设计者在选择处理器时要考虑的主要因素有以下 3 个方面。

(1) 具体的应用类型

面向应用是嵌入式系统的特色，具体的应用需求决定着应选择的嵌入式处理器的类型，不同的应用领域所需的处理器类型是不同的。当今嵌入式处理器发展的一个主要趋势是面向不同行业应用的特点进行开发，并且多采用 SoC 技术。因此，根据应用的类型选择合适的处理器不仅是必须的，而且也是可能的。

(2) 处理器性能和技术指标

开发人员通过应用需求分析获取了产品的功能性和非功能性指标后，分析研究市场上各大厂商提供的各款嵌入式处理器的性能指标(如功耗、体积、成本、可靠性、速度、处理能力、电磁兼容性等)，以选择满足应用需求的嵌入式处理器。选择嵌入式处理器的基本原则是满足具体功能性和非功能性指标需求、市场应用反应良好、硬件配置最少。

(3) 其他因素

还要考虑其他一些因素，比如：处理器是否有较好的软件开发工具支持、是否内置调试工具、供应商是否提供评估板以及开发人员对此系列处理器的熟悉程度等。

3. 操作系统的选择

简单的嵌入式产品开发不需要操作系统的支持，但对于复杂应用就需要实时操作系统(RTOS)的支持了。使用 RTOS 可以将复杂问题分解，减少开发人员的劳动量，提高产品的可靠性，加快产品的上市时间。由于具体嵌入式应用的功能需求存在差异以及众多的 RTOS 间具有不同的性能指标，因此，选择 RTOS 时也有许多因素要考虑：首先是它们的性能评价指标；其次，要考虑是选用商用的还是免费的；此外，要考虑支持何种处理器硬件平台以及何种 API，是否支持内存管理单元 MMU，还要考虑可移植性、对调试的支持情况、对标准的支持情况等。如果开发网络应用，还需要考虑该 RTOS 是否支持 TCP/IP 的网络组件和 I/O 服务等。如果开发游戏和娱乐市场，要着重考察该 RTOS 对多媒体的支持能力。

4. 嵌入式系统的交叉开发环境

嵌入式系统通常是一个资源受限的系统，因此直接在嵌入式系统的硬件平台上编写软件比较困难，甚至是不可能的。因此，需要一个交叉开发环境(Cross Development Environment，CDE)。所谓交叉开发是指在通用计算机(如通用 PC)上编辑、编译程序，生成目标平台上可以运行的二进制代码格式指令，最后再下载到目标平台上运行调试的开发方式，如图 1-3 所示。

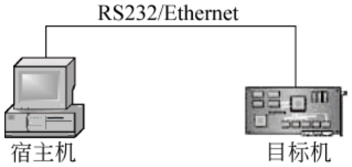


图 1-3 宿主机/目标机调试模式

通用计算机一般称为宿主机，目标平台称为目标机。交叉开发环境一般由运行于宿主机上的交叉开发软件和系统仿真器组成，交叉开发软件一般为一个整合了编辑、交叉编译/汇编、链接、交叉调试、工程管理及函数库等功能模块的集成开发环境(Integrated Development Environment，IDE)。

交叉编译器用于在宿主机上生成能在目标机上运行的代码，而交叉调试器和系统仿真器则用于在宿主机与目标机间完成嵌入式软件的调试。在采用宿主机/目标机模式开发嵌入式应用软件时，首先利用宿主机上丰富的资源和良好的开发环境开发和仿真调试目标机上的软件；然后通过串口或者以太网将交叉编译生成的目标代码传输并装载到目标机上，并在监控程序或者操作系统的支持下利用交叉调试器进行分析和调试；最后，目标机在特定环

境下脱离宿主机单独运行。

5. 嵌入式系统的调试

调试是嵌入式系统开发过程的重要环节。在嵌入式系统中，调试器是运行在宿主机操作系统上的应用程序，被调试程序是运行在目标机操作系统上的应用程序，两个程序间需要实时通信。调试嵌入式系统时，主机上运行的集成开发调试工具(调试器)通过仿真器和目标机相连。仿真器用于处理宿主机和目标机之间的所有通信，这个通信口可以是串口、并行口或者高速以太网接口。仿真器通过 JTAG 口或其他接口与目标机相连。嵌入式系统开发过程中的调试方式有很多种，应根据实际的开发要求和条件进行选择，以下是几种常用的调试方式。

(1) 源程序模拟器

源程序模拟器(例如 ARM 公司的 ARMulator)是一种利用宿主机端的软件模拟目标机的环境来执行目标机源程序的模拟调试方法，这样做的好处是在目标机硬件环境还没有建立起来时即可用模拟环境调试程序。需要注意的是，模拟器毕竟是以一种处理器模拟另一种处理器运行的，在指令执行时间、中断响应、定时器等方面很可能与实际处理有相当大的差别，它无法和 ICE 一样，仿真嵌入式系统在应用系统中的实际执行情况。

(2) ROM 监控器

ROM 监控器(例如 ARM 公司的 Angel)是一段运行在目标 ROM 上的可执行程序，PC 端调试软件可通过并口、串口、网口与之交互。在使用这种调试方式时，被调试程序首先通过 ROM 监控器下载到目标机中；然后在 ROM 监控器的监控下完成调试，目前使用的绝大部分 ROM 监控器都能够完成设置断点、单步执行、查看寄存器、修改内存空间等各项调试功能。

(3) 在线仿真器

采用 ICE 方式进行交叉调试时需要使用在线仿真器(In-circuit Emulator，ICE)。在线仿真器使用仿真头代替目标板上的 CPU，可以完全仿真处理器芯片的行为，并且提供了非常丰富的调试功能，但结构较复杂、价格昂贵，通常用于硬件开发中。

(4) 在线调试器

采用 ICD 方式进行交叉调试时需要使用在线调试器(In-circuit Debugger，ICD)。由于 ICE 的价格非常昂贵，并且每种 CPU 都需要一种与之对应的 ICE，使得开发成本非常高。一个比较好的解决办法是直接在 CPU 内部实现调试功能，并通过在开发板上引出的调试端口，发送调试命令和接收调试信息，完成调试过程。目前 Motorola 公司提供的开发板上使用的是 BDM(Background Debug Monitor)调试端口，而 ARM 公司提供的开发板上使用的则是 JTAG(Joint Test Action Group)调试端口，使用合适的软件工具与这些调试端口进行连接，可以获得与 ICE 类似的调试效果。

1.3.4 嵌入式操作系统

任务： 掌握嵌入式操作系统的概念、特点及作用。

由于硬件的限制，在使用 MCU 设计嵌入式系统的时代初期，程序设计人员得到的只有硬件系统的“裸机”，没有任何类似于操作系统的软件作为开发平台，对 CPU、RAM 等这些

硬件资源的管理工作必须由程序员自己编写程序来完成。现在，随着软、硬件技术的发展，在嵌入式系统中使用操作系统已成为一种趋势，这种运行在嵌入式硬件平台上，对整个系统及其部件、装置等资源进行统一协调、指挥和控制的系统软件就叫做嵌入式操作系统。

在一般情况下，嵌入式操作系统可以分为两类，一类是面向控制、通信等领域的实时操作系统，如 Windriver 公司的 VxWorks、ISI 的 pSOS、QNX 系统软件公司的 QNX、ATI 的 nucleus 等；另一类是面向消费电子产品的非实时操作系统，这类产品包括个人数字助理（PDA）、移动电话、机顶盒、电子书、WebPhone 等。嵌入式操作系统具有以下特点。

（1）可装卸性。开放性、可伸缩性的体系结构。

（2）强实时性。EOS 实时性一般较强，可用于各种设备控制当中。

（3）统一的接口。提供各种设备驱动接口。

（4）操作方便、简单，提供友好的图形 GUI，追求易学易用。

（5）提供强大的网络功能，支持 TCP/IP 及其他协议，提供 TCP/UDP/IP/PPP 支持及统一的 MAC 访问层接口，为各种移动计算设备预留接口。

（6）具有强稳定性和弱交互性。嵌入式系统一旦开始运行就不需要用户过多的干预，这就要求负责系统管理的 EOS 具有较强的稳定性。嵌入式操作系统的用户接口一般不提供操作命令，它通过系统调用命令向用户程序提供服务。

（7）固化代码。嵌入式操作系统和应用软件被固化在嵌入式系统计算机的 ROM 中。

（8）良好的移植性。为了适应多种多样的硬件平台，嵌入式操作系统应该可以在不做大量修改的情况下稳定地运行于不同的平台上。

目前，常用的嵌入式操作系统包括 Windows CE、VxWorks、pSOS、QNX、Palm OS、嵌入式 Linux 等。

小结

本章介绍了嵌入式系统的概况，包括嵌入式系统的定义、发展阶段、系统构成，之后分别介绍了嵌入式硬件与软件系统，需要着重区分通用计算机和嵌入式系统软件开发存在的不同之处，为之后的学习打下基础。

第2章

ARM微处理器

学习目标

通过本章的学习，应该掌握：

✍ ARM 微处理器的应用与选型

✍ ARM 微处理器的数据类型和工作状态

✍ ARM 微处理器的工作模式

✍ ARM 微处理器的寄存器

2.1 ARM微处理器概述

问题：ARM 微处理器有哪些技术特点，其应用在哪些领域？
重点：ARM 微处理器的技术特点。
内容：ARM 微处理器的技术特点和其应用领域。

ARM(Advanced RISC Machine)公司 1991 年成立于英国剑桥，主要设计 ARM 系列 RISC 处理器内核，并出售 ARM 内核设计技术的授权给生产和销售半导体的合作伙伴。另外 ARM 公司也提供了基于 ARM 架构的开发设计技术，如软件工具、评估板、调试工具、应用软件、总线架构、外围设备单元等。

ARM 公司作为知识产权供应商，本身不直接从事芯片生产，靠转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 微处理器芯片进入市场。目前，Motorola、IBM、TI、Philips、Hynix、Atmel 和 Samsung 等几十家大的半导体公司都获得了 ARM 公司的授权，生产形态各异的 ARM 芯片。这些半导体公司的加入使得 ARM 技术获得更多的第三方工具、制造、软件的支持，又使整个系统的成本降低，产品进入市场容易被消费者所接受，具有更高的竞争力。

20 世纪 90 年代以来，ARM 32 位嵌入式微处理器的应用扩展到世界范围，占据了低功耗、低成本和高性能的嵌入式系统应用领域的领先地位，形成了 32 位 RISC 微处理器的实际标准，因此 ARM 既可以看做是一个公司的名字，也可以看做是对一类微处理器的通称，还可以看做是一种技术的名字。

2.1.1 ARM微处理器的技术特点

任务：了解 ARM 微处理器的技术特点。

ARM 微处理器芯片采用 32 位 RISC 架构，具有如下特点。

(1) 支持 Thumb(16 位)/ARM(32 位)双指令集，能很好地兼容 8 位/16 位器件。

(2) 大量使用寄存器，大多数数据操作都在寄存器中完成。

(3) 寻址方式灵活简单，执行效率高。

(4) 通过专用的载入和存储指令访问存储器。

(5) 指令长度固定。

此外，ARM 体系也采用了一些别的技术，在保证高性能的基础上尽量减小芯片面积，降低芯片功耗。这些技术如下。

(1) 为提高指令的执行效率，所有的指令都可以按条件执行。

(2) 同一条数据处理指令中可包含算术逻辑单元处理和移位处理。

(3) 使用地址自动增加(减少)方法来优化程序中的循环处理。

(4) 载入和存储指令可以批量传输数据，从而提高数据传输效率。

2.1.2 ARM微处理器的应用领域

任务：了解 ARM 微处理器的应用领域。

目前，采用 ARM 技术知识产权(Intellectual Property，IP)核的微处理器，即通常所说的 ARM 微处理器，已遍及工业控制、消费类电子产品、成像和安全产品、网络系统、无线通信等领域市场，基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 75%以上的市场份额，ARM 技术正逐步渗入人们生活的各个方面。

(1) 工业控制领域：作为 32 位的 RISC 架构，基于 ARM 核的微控制器芯片不但占据了高端微控制器市场的大部分市场份额，同时也逐渐向低端微控制器应用领域扩展，ARM 微控制器的低功耗、高性价比向传统的 8 位/16 位微控制器提出了挑战。

(2) 无线通信领域：目前已有超过 85%的无线通信设备采用了 ARM 技术，ARM 以其高性能和低成本，在该领域的地位日益巩固。

(3) 网络系统：随着宽带技术的推广，采用 ARM 技术的 ADSL 芯片正逐步获得竞争优势。此外，ARM 在语音及视频处理上进行了优化，并获得广泛支持，也对 DSP 的应用领域提出了挑战。

(4) 消费类电子产品：ARM 技术在目前流行的数字音频播放器、数字机顶盒和游戏机中得到广泛采用。

(5) 成像和安全产品：现在流行的绝大部分数字照相机和打印机中都采用 ARM 技术。手机中的 32 位 SIM 智能卡也采用了 ARM 技术。

除此以外，ARM 微处理器及其技术还应用到许多不同的领域中，并会在将来获得更加广泛的应用。

2.2 ARM微处理器体系结构

问题：什么是 RISC？什么是 CISC？ARM 体系结构目前有哪些版本，有哪些变种？ARM 微处理器目前包括哪几个系列，每个系列有哪些特点？

重点：RISC 和 CISC 的定义，ARM 体系结构版本和目前的 ARM 微处理器系列及特点。

内容：RISC 体系结构、ARM 版本及其变种和 ARM 微处理器系列及特点。

2.2.1 RISC 体系结构

任务：掌握 RISC 和 CISC 体系结构的含义，RISC 体系结构的优点。

提到 ARM 就不得不提计算机体系的两个相对的概念：RISC 和 CISC。

CISC(Complex Instruction Set Computer，复杂指令集计算机)是传统的计算机体系结构，在 20 世纪 90 年代前被广泛使用，其特点是通过存放在只读存储器中的微码来控制整个处理器的运行。一条指令的执行往往可以完成一系列运算，但需要耗费多个时钟周期。随着需求的不断增加，设计的指令集越来越多，为支持这些新增的指令，计算机的体系结构会

越来越复杂。并且各种指令的使用频度都不太高且差别很大,大约有 20%的指令会被反复使用,占整个程序运行时间的 80%;而余下的 80%的指令却不经常使用,在程序设计中仅占 20%。这就是所谓的 20%~80%定律。

针对 CISC 结构存在的这些问题,Patterson 等人提出了精简指令集计算机(RISC-Reduced Instruction Set Computer)的设想。通过精简指令来使计算机结构变得简单、合理、有效,并克服 CISC 结构的上述缺点。

RISC 是一种设计思想,并不是一种产品,它还在不断地发展和丰富,它是近代计算机体系结构发展的一个里程碑。CMU 发表的学术论文提出了设计 RISC 机器应当遵循的一般原则,包括如下几个。

(1) 确定指令系统时,只选择使用频度很高的那些指令,在此基础上增加少量能有效支持操作系统和高级语言实现及其他功能的最有用的指令,使指令的条数大大减少,一般不超过 100 条。

(2) 大大减少指令系统可采用的寻址方式的种类,一般不超过两种。简化指令的格式,将其也限制在两种之内,并使全部指令都具有相同的长度。

(3) 使所有指令都在一个机器周期内完成。

(4) 扩大通用寄存器的个数,一般不少于 32 个寄存器,以尽可能减少访问内存的操作,所有指令中只有存(STORE)、取(LOAD)指令才可以访问内存,其他指令的操作一律都在寄存器间进行。

(5) 为提高指令执行速度,大多数指令都采用硬联控制实现,少数指令采用微程序实现。

(6) 通过精简指令和优化编译程序设计,以简单有效的方式来支持高级语言的实现。

总之,减小指令平均周期是 RISC 设计思想的精华。

从 1980 年以来,所有新的处理器体系结构都或多或少地采用了 RISC 的概念,甚至有些典型的 CISC 处理机中也采用了 RISC 设计思想,比如 Intel 公司的 80486、Pentium 系列等。而应用 RISC 思想最成功也是第一个商业化的实例就是 ARM 微处理器,当然它也放弃了一些 RISC 特征而保留了一些 CISC 特征。

2.2.2　ARM 体系结构版本

任务: 了解 ARM 体系结构的各个版本及其特点。

ARM 指令集体系结构从最初开发至今已有了重大改进,而且将会不断完善和发展。为了精确表达每个 ARM 实现中所使用的指令集,到目前 ARM 体系结构共定义了 6 个版本,各版本的特点如下。

1. Version 1(v1)

该版本包括如下内容。

(1) 基本数据处理指令(不包括乘法指令)。

(2) 字节、字以及半字加载和存储。

(3) 软件中断指令。

(4) 分支指令。

(5) 26 位地址总线。

2. Version 2(v2)

该版本增加了下列指令。

(1) 乘法和乘加指令(Multiply & Multiply-accumulate)。

(2) 支持协处理器的指令。

(3) 对于 FIQ 模式提供了额外的两个备份寄存器。

(4) SWP 指令及 SWPB 指令。

(5) 26 位地址总线。

3. Version 3(v3)

该版本推出 32 位寻址能力,主要结构扩展变化如下。

(1) 32 位地址总线,但除版本 3G(版本 3 的一个变种)外其他版本都是向前兼容的,支持 26 位地址总线。

(2) 当前程序状态信息从原来的 R15 移到一个新的寄存器——当前程序状态寄存器(Current Program Status Register,CPSR)中。

(3) 增加了备份程序状态寄存器(Saved Program Status Register,SPSR),用于在程序异常中断程序时,保存被中断程序的程序状态。

(4) 增加了两种处理器模式,使操作系统代码可以方便地使用数据访问中止异常、指令预取中止异常和未定义指令异常。

(5) 增加了指令 MSR 和 MRS,用于访问 CPSR 和 SPSR。

(6) 增加了从异常返回的指令。

4. Version 4(v4)

该版本增加了下列指令。

(1) 半字加载和存储指令。

(2) 加载带符号的字节和半字数据的指令。

(3) 增加了 T 变种,可以将处理器状态切换到 Thumb 状态。

(4) 增加了处理器的特权模式。

该版本不再强制要求与以前的 26 位地址空间兼容。

5. Version 5(v5)

该版本改进了 ARM 和 Thumb 之间的交互,增加或修改了下列指令。

(1) 增加了前导零计数(Count Leading Zeros,CLZ)指令,该指令可以使整数除法和中断优先级排队操作更为有效。

(2) 增加了软件断点指令。

(3) 更加严格地定义了乘法指令对条件标志位的影响。

6. Version 6(v6)

该版本主要是增加了 SIMD 功能扩展。它适用于使用电池供电的高性能的便携式设备。这些设备一方面需要处理器提供高性能,另一方面又需要功耗很低。SIMD 功能扩展为包括音频/视频处理在内的应用提供了优化功能。它可以使音频/视频处理性能提高 4 倍。Version 6 首先在 2002 年春季发布的 ARM11 处理器中使用。

2.2.3 ARM 体系结构的变种及版本命名格式

任务：了解 ARM 体系结构各变种的特点及版本命名格式。

1. ARM 体系结构的变种

通常将某些特定功能称为 ARM 体系的某种变种，例如支持 Thumb 指令集的 ARM 体系称为 T 变种，到目前为止，ARM 定义了下面一些变种。

(1) T 变种

T 变种是支持 Thumb 指令集的 ARM 体系。Thumb 指令集是将 32 位 ARM 指令集的一个子集重新编码而形成的一个指令集。Thumb 指令的长度是 16 位。Thumb 指令集可以得到比 ARM 指令集密度更高的代码，能够以 16 位的指令实现 32 位的指令功能，这对需要严格控制产品成本的设计是非常有意义的。目前，Thumb 指令集有两个版本：Thumb-1 和 Thumb-2。Thumb-1 是 ARM 体系版本 4 的 T 变种，Thumb-2 是 ARM 体系版本 5 的 T 变种。

(2) M 变种(长乘法指令)

长乘法指令是一种生成 64 位乘法结果的乘法指令。M 变种增加了两条用于进行长乘法操作的 ARM 指令：一条用于完成 32 位整数乘以 32 位整数生成 64 位整数结果的长乘法操作；另一条用于完成 32 位整数乘以 32 位整数，再加上 64 位整数生成 64 位整数结果的长乘加法操作。

对于支持长乘法指令的 ARM 体系版本，用字母 M 来表示。M 变种首先在 ARM 体系版本 3 中引入，在 ARM 体系版本 4 及其以后的版本中，M 变种是系统的标准部分，所以字母 M 通常也不单独列出来。

(3) E 变种

E 变种增加了一些附加指令用于增强处理器对一些典型的 DSP 算法的处理性能，具体如下。

① 几条新的实现 16 位数据乘法和乘加操作的指令。

② 实现饱和的带符号数的加减法操作的指令。所谓饱和的带符号数的加减法操作是在加减法操作溢出时，结果并不进行卷绕，而是使用最大的整数或最小的负数来表示。

③ 进行双字数据处理的指令，包括双字加载指令 LDRD、双字存储指令 STRD 和协处理器的寄存器传输指令 MCRR/MRRC。

④ cache 预取指令 PLD。

E 变种首先在 ARM 体系版本 5 中引入，用字母 E 来表示。

(4) J 变种(Java 加速器 Jazelle)

ARM 的 Jazelle 技术将 Java 的优势和先进的 32 位 RISC 芯片完美地结合在一起。Jazelle 技术提供了 Java 加速功能，可以得到比普通 Java 虚拟机高得多的性能。与普通的 Java 虚拟机相比，Jazelle 使 Java 代码的运行速度提高了 8 倍，而功耗降低了 80%。

Jazelle 技术使得程序员可以在一个独立的处理器上同时运行 Java 应用程序、已经建立好的操作系统、中间件以及其他的应用程序。与使用协处理器和双处理器相比，使用单独的处理器可以在提供高性能的同时保证低功耗和低成本。

J 变种首先在 ARM 体系版本 4 中使用，用字母 J 表示。

(5) SIMD 变种

ARM 媒体功能扩展 SIMD 技术极大地提高了嵌入式应用系统的音频和视频处理器的能力，它可以使微处理器的音频和视频性能提高 4 倍。新一代的 Internet 应用产品、移动电话和 PDA 等设备终端需要提供高性能的流式媒体，包括音频和视频等，而且这些设备需要提供更加人性化的界面，包括语音输入和手写输入等。这样，就对处理器的数字信号处理能力提出了很高的要求，同时还必须保证低功耗。ARM 的 SIMD 媒体功能扩展为这些应用系统提供了解决方案，它为包括音频和视频处理在内的应用系统提供了优化功能，其主要特点如下。

① 使处理器的音频和视频性能提高了 2～4 倍。

② 可同时进行 2 个 16 位操作数或者 4 个 8 位操作数的运算。

③ 用户可自定义饱和运算的模式。

④ 可进行 2 个 16 位操作数的乘加/乘减运算及 2 个 32 位小数的乘加运算。

⑤ 同时进行 8/16 位选择操作。

2. ARM 体系结构版本的命名格式

ARM 体系结构版本的命名包括以下几个部分。

(1) 基本字符串 ARMv。

(2) 基本字符串 ARMv 后是 ARM 指令集版本号，目前是数字 1～6。

(3) ARM 指令集版本号后是表示所含变种的字符。

(4) 最后的字符 x 表示排除某种功能。例如，在早期的一些 E 变种中，未包含双字加载指令 LDRD、双字存储指令 STRD、协处理器的寄存器传输指令 MCRR/MRRC 以及 cache 预取指令 PLD。这种 E 变种记作 ExP，其中 x 表示缺少，P 代表上述的几种指令。

例如，ARMv5TExP 表示 ARM 体系结构的版本 5，含 T 变种、M 变种，未包含 P。

2.2.4 ARM 微处理器系列

任务：了解每一个 ARM 微处理器系列的特点和应用领域。

ARM 微处理器目前包括下面几个系列，以及其他厂商基于 ARM 体系结构的处理器，除了具有 ARM 体系结构的共同特点以外，每一个系列的 ARM 微处理器都有各自的特点和应用领域。

(1) ARM 7 系列。

(2) ARM 9 系列。

(3) ARM 9E 系列。

(4) ARM 10E 系列。

(5) SecurCore 系列。

(6) Intel 的 Xscale。

(7) Intel 的 StrongARM。

其中，ARM 7、ARM 9、ARM 9E 和 ARM 10E 为 4 个通用处理器系列，每一个系列提供一套相对独特的性能来满足不同应用领域的需求。SecurCore 系列专门为安全要求较高

的应用而设计。

1. ARM 7 微处理器系列

ARM 7 系列微处理器为低功耗的 32 位 RISC 处理器，最适合用于对价位和功耗要求较高的消费类应用，如数字移动电话。ARM 7 微处理器系列具有如下特点。

(1) 具有嵌入式 ICE-RT 逻辑，调试开发方便。

(2) 极低的功耗，适合对功耗要求较高的应用，如便携式产品。

(3) 能够提供 0.9MIPS/MHz 的三级流水线结构。

(4) 代码密度高并兼容 16 位的 Thumb 指令集。

(5) 对操作系统的支持广泛，包括 Windows CE、Linux、Palm OS 等。

(6) 指令系统与 ARM 9 系列、ARM 9E 系列和 ARM 10E 系列兼容，便于用户的产品升级换代。

(7) 主频最高可达 130MIPS，高速的运算处理能力能满足绝大多数复杂应用的要求。

ARM 7 系列微处理器的主要应用领域为：工业控制、Internet 设备、网络和调制解调器设备、移动电话等多种多媒体和嵌入式应用。

ARM 7 系列微处理器包括如下几种类型的核：ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ。其中，ARM7TDMI 是目前低端的 32 位嵌入式 RISC 处理器，使用广泛。

ARM7TDMI 的 TDMI 的基本含义如下。

(1) T：支持 16 位压缩指令集 Thumb。

(2) D：支持片上 Debug，使处理器能够停止以响应调试请求。

(3) M：内嵌硬件乘法器(multiplier)。

(4) I：嵌入式 ICE，支持片上断点和调试点。

2. ARM 9 微处理器系列

ARM 9 系列有 ARM9TDMI 内核及在此基础上发展起来的 ARM920T、ARM922T、ARM940T 等内核，以适用于不同的应用场合。所有的 ARM 9 系列微处理器都带有 Thumb 指令集和基于嵌入式 ICE JTAG 的软件调试方式。ARM 9 系列兼容 ARM 7 系列，而且具有比 ARM 7 更加灵活的设计。ARM 9 系列微处理器在高性能和低功耗特性方面提供最佳的性能，具有以下特点。

(1) 5 级整数流水线，指令执行效率更高。

(2) 提供 1.1MIPS/MHz 的哈佛结构。

(3) 支持 32 位 ARM 指令集和 16 位 Thumb 指令集。

(4) 支持 32 位的高速 AMBA 总线接口。

(5) 全性能的 MMU，支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。

(6) MPU 支持实时操作系统。

(7) 支持数据 cache 和指令 cache，具有更高的指令和数据处理能力。

ARM 9 系列微处理器采用 ARMv4T 哈佛结构，5 级流水处理以及分离的 cache 结构，平均功耗为 0.7mW/MHz，时钟频率为 120～200MHz，每条指令平均执行 1.5 个时钟周期。与 ARM 7 系列相似，其中的 ARM920T 和 ARM940T 为含 cache 的 CPU 核。

ARM 9 系列微处理器主要应用于无线设备、仪器仪表、安全系统、机顶盒、高端打印机、

数字照相机和数字摄像机等。

3. ARM 9E 微处理器系列

ARM 9E 系列微处理器为可综合处理器,使用单一的处理器内核提供了微控制器、DSP、Java 应用系统的解决方案,极大地减小了芯片的面积和系统的复杂程度。ARM 9E 系列微处理器提供了增强的 DSP 处理能力,很适合于那些需要同时使用 DSP 和微控制器的应用场合。

ARM 9E 系列微处理器的主要特点如下。

(1) 支持 DSP 指令集,适合于需要高速数字信号处理的场合。

(2) 5 级整数流水线,指令执行效率更高。

(3) 支持 32 位 ARM 指令集和 16 位 Thumb 指令集。

(4) 支持 32 位的高速 AMBA 总线接口。

(5) 支持 VFP9 浮点处理协处理器。

(6) 全性能的 MMU,支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。

(7) MPU 支持实时操作系统。

(8) 支持数据 cache 和指令 cache,具有更强大的指令和数据处理能力。

(9) 主频最高可达 300MIps。

ARM 9E 系列微处理器广泛应用于硬盘驱动器和 DVD 播放器等海量存储设备、语音编码器、免提链接、反锁刹车等自动控制解决方案以及调制解调器、语音识别和合成等设备中。

ARM 9E 系列微处理器包含 ARM926EJ-S、ARM946E-S 和 ARM966E-S 共 3 种类型,分别用于不同的应用场合。

4. ARM 10E 微处理器系列

ARM 10E 系列微处理器具有高性能、低功耗的特点。由于采用了新的体系结构,ARM 10E 系列微处理器在所有的 ARM 产品中具有最高的指令执行速度和系统主频。ARM 10E 系列微处理器采用了先进的节能方式,使其功耗极低。同时,ARM 10E 系列微处理器提供了 64 位的加载/存储体系,支持包括向量操作的满足 IEEE 754 的浮点运算协处理器,使系统集成更加方便,拥有完整的硬件和软件开发工具。

ARM 10E 系列微处理器的主要特点如下。

(1) 支持 DSP 指令集,适合于需要高速数字信号处理的场合。

(2) 6 级整数流水线,指令执行效率更高。

(3) 支持 32 位 ARM 指令集和 16 位 Thumb 指令集。

(4) 支持 32 位的高速 AMBA 总线接口。

(5) 支持 VFP10 浮点处理协处理器。

(6) 全性能的 MMU,支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。

(7) 支持数据 cache 和指令 cache,具有更强大的指令和数据处理能力。

(8) 主频最高可达 400MIps。

(9) 内嵌并行读/写操作部件。

ARM 10E 系列微处理器主要应用于下一代无线设备、数字消费品、成像设备、工业控制、汽车、通信和信息系统等领域。

ARM 10E 系列微处理器包含 ARM1020E、ARM1022E 和 ARM1026EJ-S 共 3 种类型,分别用于不同的应用场合。

5. SecurCore 系列微处理器

SecurCore 系列微处理器提供了完善的 32 位 RISC 技术的安全解决方案，专为安全需要而设计。SecurCore 系列微处理器除了具有 ARM 体系结构的体积小、功耗低、性能高等优点外，还具有其独特的优势，即提供了对安全解决方案的支持。

SecurCore 系列微处理器具有如下特点。

(1) 支持 32 位 ARM 指令集和 16 位 Thumb 指令集。

(2) 提供面向智能卡的和低成本的存储保护单元。

(3) 带有灵活的保护单元，以确保操作系统和应用数据的安全。

(4) 采用软内核技术，防止外部对其进行扫描探测。

(5) 可集成用户自己的安全特性和其他协处理器。

SecurCore 系列微处理器主要应用于一些对安全性要求较高的应用产品及应用系统中，如电子商务、电子政务、电子银行业务、网络和认证系统等领域。

SecurCore 系列微处理器包含 SecurCore SC100、SecurCore SC110、SecurCore SC200 和 SecurCore SC210 共 4 种类型，分别用于不同的应用场合。

6. StrongARM 系列微处理器

StrongARM 系列微处理器是 Intel 生产的采用 ARM 体系结构、高度集成的 32 位 RISC 微处理器，实现了 ARMv4 体系结构。它采用先进的 CMOS 工艺、先进的流水线设计、精密的时钟分配方案和功耗设计，具有较高的性能和非常低的功耗。StrongARM 系列微处理器具有以下特点。

(1) 支持 32 位 ARM 指令集和 16 位 Thumb 指令集。

(2) 实现了 ARMv4 体系结构，并且增加了 cache 容量，支持虚拟存储器管理。

(3) 协处理器 CP15 包含一些寄存器，用于控制和配置 cache、写缓存、虚拟存储器管理 MMU、读缓存、断点以及一些时钟功能等。

(4) 具有 MMU 部件，用于将虚拟地址转换成物理地址和控制存储器访问权限。

(5) 虚拟存储器管理部件分为指令存储器管理单元和数据存储器管理单元。

Intel StrongARM 微处理器是便携式通信产品和消费类电子产品的理想选择，已成功应用于多家公司的掌上电脑系列产品中。

7. Intel 的 Xscale 系列微处理器

Xscale 微处理器是采用 Intel Pentium 技术实现的、与 ARMv5TE 兼容的嵌入式微处理器架构。它对 ARM 体系结构进行了增强，其主频可以超出普通 ARM 微处理器的主频数倍，高达 1GHz 以上，具有业界领先的高性能、低功耗和高性价比的处理器。它的设计目标是“面向特定应用的标准产品”，已使用在数字移动电话、个人数字助理和网络产品等场合。

Xscale 微处理器实现了 ARMv5TE 整数指令集体系结构，包括 ARM 体系结构的 32 位 ARM 指令集、16 位的 Thumb 指令集和 DSP 指令集。但出于成本考虑，Xscale 微处理器没有实现 ARM 体系结构定义的浮点部件和浮点指令。Xscale 微处理器架构经过专门设计，处理速度是 Intel StrongARM 处理速度的两倍，其内部结构也有了相应的变化。

(1) 数据 cache 和指令 cache 均达到 32KB。

(2) 微小数据 cache 的容量达到 2KB。

(3) 具有 7 级超级流水线。

(4) 新增乘法/加法器 MAC 和特定的 DSP 型协处理器 CPO,以加强对多媒体技术的支持。

(5) 动态电源管理,使 Xscale 微处理器的时钟频率可达 1GHz,功耗达 1.6W,并能达到 1200MIps。

(6) 超低功耗与良好性能的组合使 Intel Xscale 微处理器被广泛用于因特网的接入设备。

2.3 ARM微处理器的编程模型

问题:ARM 微处理器的数据类型有哪些? ARM 微处理器的工作状态有哪些,怎样切换? ARM 微处理器的工作模式有哪些? ARM 微处理器的寄存器是怎样组织的?
重点:ARM 微处理器的工作状态、工作模式和寄存器的组织。
内容:ARM 微处理器的数据类型、工作状态、工作模式和寄存器的组织。

2.3.1 ARM微处理器的数据类型

任务:掌握 ARM 微处理器的数据类型有哪些。

ARM 处理器支持以下 6 种数据类型(较早的 ARM 处理器不支持半字和有符号字节)。

(1) 8 位有符号和无符号字节。

(2) 16 位有符号和无符号半字,它们以 2 字节的边界对齐。

(3) 32 位有符号和无符号字,它们以 4 字节的边界对齐。

ARM 指令全是 32 位的字,并且必须以字为单位边界对齐。Thumb 指令是 16 位半字,必须以 2 字节为单位边界对齐。

在 ARM 处理器内部,所有 ARM 操作都面向 32 位的操作数,只有数据传送指令支持较短的字节和半字的数据类型。当从存储器调入一个字节或半字时,根据指令对数据的操作类型,需要进行数据位数的扩展:将其扩展出的高位用 0(无符号数)或符号位(有符号数)填充,形成 32 位,进而作为 32 位数据在内部进行处理。

ARM 协处理器可以支持其他数据类型,特别是定义了一些表示浮点数的数据类型。在 ARM 核内没有明确支持这些数据类型,然而在没有浮点协处理器的情况下,这些类型可由软件用上述标准类型解释。

2.3.2 ARM微处理器的工作状态

任务:了解 ARM 微处理器的两种工作状态:ARM 和 Thumb,了解 ARM 和 Thumb 的切换方式。

从编程的角度看,ARM 微处理器的工作状态一般有两种:一种为 ARM 状态,此时处理器执行 32 位的字对齐的 ARM 指令;另一种为 Thumb 状态,此时处理器执行 16 位的、半

字对齐的 Thumb 指令。

当 ARM 微处理器执行 32 位的 ARM 指令集时，工作在 ARM 状态；当 ARM 微处理器执行 16 位的 Thumb 指令集时，工作在 Thumb 状态。

ARM 微处理器的两种工作状态可在程序的执行过程中进行切换，并且，微处理器工作状态的转变并不影响微处理器的工作模式和相应寄存器中的内容。

ARM 指令集和 Thumb 指令集均有指令实现微处理器两种工作状态间的切换，但 ARM 微处理器在开始执行代码时，应该处于 ARM 状态。状态切换的方法如下。

(1) 进入 Thumb 状态：当操作数寄存器的状态位(位 0)为 1 时，可以采用执行 BX 指令的方法，使微处理器从 ARM 状态切换到 Thumb 状态。此外，当处理器从 Thumb 状态进入异常(如 FIQ、IRQ、Undef、Abort、SWI 等)处理，在异常处理返回时，自动切换回 Thumb 状态。

(2) 进入 ARM 状态：当操作数寄存器的状态位为 0 时，执行 BX 指令时可以使微处理器从 Thumb 状态切换到 ARM 状态。所有的异常处理都在 ARM 状态下执行。此外，在处理器进行异常处理时，把 PC 指针放入异常模式链接寄存器中，并从异常向量地址开始执行程序，也可以使处理器切换到 ARM 状态。

范例：从 ARM 状态切换到 Thumb 状态

```
LDR R0,=LABEL+1                    ;因为 LABEL 对应的地址是 4 字节对齐的,即最低位 (位 0)为 0
BX R0                              ;将该地址加 1 后复制给 R0,R0 最低位 (位 0)是 1
```

从 Thumb 状态切换到 ARM 状态

```
LDR R0,=LABEL                      ;LABEL 对应的地址最低位 (位 0)为 0,将该地址复制给 R0
BX R0                              ;R0 最低位 (位 0)是 0
```

2.3.3　ARM 微处理器的工作模式

任务：理解 ARM 微处理器的 7 种工作模式。

ARM 微处理器有以下 7 种工作模式。

(1) 用户模式(User)：ARM 微处理器的正常运行模式，通常用来执行一般的应用程序。

(2) 快速中断模式(FIQ)：响应快速中断时的模式，由外部触发 FIQ 引脚，用于高速数据传输或通道处理。

(3) 外部中断模式(IRQ)：响应一般的外部中断时的模式，由外部触发 IRQ 引脚。

(4) 管理模式(SVC)：操作系统的保护模式。

(5) 中止模式(ABT)：当数据或指令预取中止时进入该模式，可用于实现虚拟存储及存储保护。

(6) 系统模式(SYS)：运行具有特权的操作系统任务。

(7) 未定义模式(UND)：当未定义的指令执行时进入该模式，可用于支持硬件协处理器软件仿真。

除用户模式以外，其余的所有 6 种模式称之为非用户模式或特权模式。ARM 内部部

分寄存器和一些片内外围设备在硬件设计上只允许(或可选为只允许)在特权模式下访问。此外,特权模式可以自由地切换到处理器模式,而用户模式不能直接切换到别的模式。

除去用户模式和系统模式以外的 5 种又称为异常模式。它们除了可以通过程序切换进入外,也可以由特定的异常进入。当特定的异常出现时,微处理器进入相应的模式。每种模式都有某些附加的寄存器,以避免异常退出时用户模式的状态不可靠。后面章节会对寄存器进行详细介绍。

至于系统模式,它与用户模式一样,不能由异常进入,而且使用与用户模式完全相同的寄存器。然而它是特权模式,不受用户模式的限制。在此模式下操作系统可方便地访问用户模式的寄存器。操作系统的一些特权任务可使用该模式访问一些受限制的资源而不必担心异常出现时任务状态变得不可靠。

2.3.4 ARM 微处理器的寄存器组织

任务:掌握 ARM 微处理器的寄存器组织。

ARM 微处理器共有 37 个寄存器。

(1) 其中 31 个为通用寄存器。这些寄存器是 32 位的,分别是 R0～R15、R8_fiq～R12_fiq、R13_svc、R14_svc、R13_abt、R14_abt、R13_und、R14_und、R13_irq、R14_irq、R13_fiq、R14_fiq。

(2) 另外 6 个为状态寄存器。状态寄存器也是 32 位的,但只使用了其中的 12 位。这些寄存器分别是 CPSR、SPSR_svc、SPSR_abt、SPSR_und、SPSR_irq、SPSR_fiq。

这些寄存器并不是在同一时间都可以被访问的,处理器的状态和工作模式决定了程序员可以访问哪些寄存器。ARM 状态下的寄存器组织如图 2-1 所示,图中的每一列都是在每种模式下可见的寄存器。

1. 通用寄存器

通用寄存器包括 R0～R15,可以分为以下 3 类。

(1) 未分组寄存器 R0～R7

R0～R7 是未分组寄存器,这意味着在所有的工作模式下,各个寄存器是唯一的,但都访问一样的 32 位物理寄存器。它们是真正的通用寄存器,未被体系结构用做特殊的用途,并且可用于任何使用通用寄存器的指令。在中断或异常处理进行工作模式转换时,由于不同的处理器工作模式均使用相同的物理寄存器,可能会造成寄存器中的数据被破坏,这一点在进行程序设计时应引起注意。

(2) 分组寄存器 R8～R14

R8～R14 是分组寄存器。对于分组寄存器,它们每一次所访问的物理寄存器与处理器当前的工作模式有关。几乎所有允许使用通用寄存器的指令都允许使用分组寄存器。

对于 R8～R12 来说,每个寄存器对应两个不同的物理寄存器,当使用 FIQ 模式时,访问寄存器 R8_fiq～R12_fiq;当使用除 FIQ 模式以外的其他模式时,访问寄存器 R8～R12。

寄存器 R8～R12 在 ARM 体系结构中没有特定的用途。不过对于那些只使用 R8～R14 就足够处理的简单中断来说,FIQ 单独使用的这些寄存器可实现快速的中断处理。

对于 R13、R14 来说,每个寄存器对应 6 个不同的物理寄存器,其中的一个由用户模式与系统模式共用,另外 5 个物理寄存器对应于其他 5 种不同的运行模式。

图 2-1 ARM 状态下的寄存器组织

可以采用以下的记号来区分不同模式下的物理寄存器：

```
R13_<mode>
R14_<mode>
```

其中，mode 为以下几种模式之一：User、FIQ、IRQ、SVC、ABT、UND。

寄存器 R13 在 ARM 指令中常用做堆栈指针，但这只是一种习惯用法，用户也可使用其他的寄存器作为堆栈指针。但在 Thumb 指令集中，某些指令要求必须使用 R13 作为堆栈指针。

每种异常模式都有自己独立的物理寄存器 R13，它通常指向由异常模式所专用的堆栈。在用户应用程序的初始化部分，一般要初始化每种异常模式下的 R13，使其指向该运行模式的栈空间。这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入 R13 所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复，采用这种方式可以保证异常发生后程序正常执行。

R14 又称为子程序链接寄存器(Subroutine Link Register)，或链接寄存器 LR。当执行 BL 子程序调用指令时，R14 中得到 R15(程序计数器 PC)的备份。在其他情况下，R14 用做通用寄存器。与 R13 类似，当发生中断或异常时，对应的 R14_svc、R14_irq、R14_fiq、R14_abt 和R14_und 用来保存 R15 的返回值。

寄存器 R14 常用在如下情况下：在每种工作模式下都可用 R14 保存子程序的返回地

址，当用 BL 或 BLX 指令调用子程序时，将 PC 的当前值复制给 R14，执行完子程序后，又将 R14 的值复制回 PC，即可完成子程序的调用返回。

（3）程序计数器 R15(PC)

寄存器 R15 用做程序计数器(PC)。在 ARM 状态下，位[1:0]为 0；在 Thumb 状态下，位[0]为 0；R15 虽然也可用做通用寄存器，但一般不这么使用，因为对 R15 的使用有一些特殊的限制，当违反了这些限制时，程序的执行结果是未知的。

由于 ARM 体系结构采用了多级流水线技术，对于 ARM 指令集而言，PC 总是指向当前指令的后面指令的地址。比如，ARM 7 采用的是 3 级流水线，那么执行 ARM 指令时 PC 的值为当前指令的地址值加 8 个字节，即指向当前指令后面的第二条指令的地址。

2. 程序状态寄存器

ARM 体系结构包含一个当前的程序状态寄存器(CPSR)和 5 个备份的程序状态寄存器(SPSR)。程序状态寄存器的每一位的安排如图 2-2 所示。备份的程序状态寄存器用来进行异常处理，其包括的功能有以下几个。

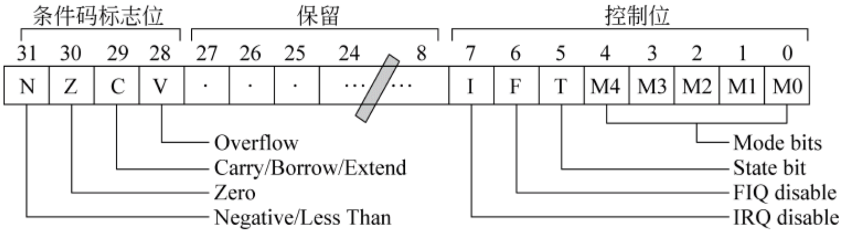


图 2-2　程序状态寄存器格式

① 保存 ALU 中的当前操作信息。

② 控制允许和禁止中断。

③ 设置处理器的工作模式。

（1）条件码标志(Condition Code Flags)

条件码标志包括 N(Negtive)、Z(Zero)、C(Carry)、V(Overflow)等标志位。它们的内容可被算术或逻辑运算的结果所改变，并且可以决定某条指令是否被执行。

条件码标志位的具体含义如表 2-1 所示。

（2）控制位

程序状态寄存器的低 8 位(包括 I、F、T 和 M[4:0])称为控制位，当发生异常时这些位可以被改变。如果处理器运行在特权模式下，这些位也可以由程序修改。

① 中断禁止位 I、F：这两位决定是否允许响应中断。

I=1 表示禁止 IRQ 中断；

F=1 表示禁止 FIQ 中断。

② 处理器状态位 T：该位反映处理器的运行状态。

对于 ARM 体系结构 v5 及以上版本的 T 系列处理器，T=0 时处理器运行于 ARM 状态下；T=1 时处理器运行于 Thumb 状态下。

表 2-1　条件码标志位的含义

标志位	含　　义
N	当用两个补码表示的带符号数进行运算时，N=1 表示运算的结果为负数；N=0 表示运算的结果为正数或 0
Z	Z=1 表示运算的结果为 0；Z=0 表示运算的结果为非 0
C	可以采用 4 种方法设置 C 的值： ① 加法运算(包括比较指令 CMN)：当运算过程中产生了进位(无符号数溢出)时，C=1，否则 C=0 ② 减法运算(包括比较指令 CMP)：当运算过程中产生了借位(无符号数溢出)时，C=0，否则 C=1 ③ 对于包含移位操作的非加/减运算指令，C 为移出值的最后一位 ④ 对于其他的非加/减运算指令，C 的值通常不受影响
V	可以采用两种方法设置 V 的值： ① 对于加/减运算指令，当操作数和运算结果为用二进制的补码表示的带符号数时，V=1 表示符号位溢出 ② 其他的指令通常不影响 V 位
Q	在 ARM v5 及以上版本的 E 系列处理器中，用 Q 标志位指示增强的 DSP 运算指令是否发生了溢出；在其他版本的处理器中，Q 标志位无定义

对于 ARM 体系结构 v5 及以上版本的非 T 系列处理器，当该位为 0 时，表示运行于 ARM 状态下；当该位为 1 时，执行下一条指令将引起未定义指令异常。

③ 运行模式位 M[4:0]：M0、M1、M2、M3、M4 是模式位。这些位决定了处理器的工作模式。具体含义如表 2-2 所示。

表 2-2　处理器的运行模式位 M[4:0]组合

M[4:0]	处理器模式	可访问的寄存器
0b10000	用户模式	PC，CPSR，R0～R14
0b10001	FIQ 模式	PC，CPSR，SPSR_fiq，R14_fiq～R8_fiq，R7～R0
0b10010	IRQ 模式	PC，CPSR，SPSR_irq，R14_irq，R13_irq，R12～R0
0b10011	管理模式	PC，CPSR，SPSR_svc，R14_svc，R13_svc，R12～R0
0b10111	中止模式	PC，CPSR，SPSR_abt，R14_abt，R13_abt，R12～R0
0b11011	未定义模式	PC，CPSR，SPSR_und，R14_und，R13_und，R12～R0
0b11111	系统模式	PC，CPSR(ARM v4 及以上版本)，R14～R0

由表 2-3 可知，并不是所有的工作模式位的组合都是有效的，其他的组合结果会导致处理器进入一个不可恢复的状态。

(3) 保留位

程序状态寄存器中的其余位为保留位，当改变程序状态寄存器中的条件标志位或者控制位时，保留位不变，在程序中也不要使用保留位来存储数据。保留位将用于 ARM 版本的扩展。

Thumb 状态下的寄存器集是 ARM 状态下的寄存器集的子集，它有 8 个通用寄存器(R0～R7)以及 PC、SP、LR 和 CPSR，在每一种特权模式下均有一组 SP、LR 和 SPSR 的物理寄存器，作用与在 ARM 状态下相同。

2.3.5 ARM 体系中的存储模式

任务：掌握 ARM 体系中的两种存储模式：大端模式和小端模式。

ARM 体系结构将存储器看做是从零地址开始的字节的线性组合，即一个字节对应一个地址，对应 32 位的字长，第一个存储字包含的地址为 0～3，第二个存储字包含的地址是4～7，依次排列。作为 32 位的微处理器，ARM 体系结构所支持的最大寻址空间为 4GB(2^{32}B)。

ARM 体系结构可以用两种方法存储字数据，分别称为大端模式和小端模式。这两种模式是根据最低有效字节与相邻较高有效字节存放在较低地址还是较高地址来划分的。

小端模式：字的高位字节对应存储在高地址，低位字节对应存储在相邻的低地址。

大端模式：字的高位字节对应存储在低地址，低位字节对应存储在相邻的高地址。

例如，在存储地址 0x30000100～0x30000103 处存储的字节数据为 0x12、0x34、0x56、0x78。若看做字，采用小端模式时该字为 0x78563412；若采用大端模式，该字为 0x12345678。对应的两种存储模式如图 2-3 所示。

小端模式

0x30000103	0x30000102	0x30000101	0x30000100
31 ～ 24	23 ～ 16	15 ～ 8	7 ～ 0
0 1 1 1 1 0 0 0	0 1 0 1 0 1 1 0	0 0 1 1 0 1 0 0	0 0 0 1 0 0 1 0

大端模式

0x30000103	0x30000102	0x30000101	0x30000100
31 ～ 24	23 ～ 16	15 ～ 8	7 ～ 0
0 0 0 1 0 0 1 0	0 0 1 1 0 1 0 0	0 1 0 1 0 1 1 0	0 1 1 1 1 0 0 0

图 2-3 两种存储模式

小端模式是 ARM 处理器的默认模式。ARM 指令集没有提供任何直接选择小端模式或大端模式的指令。取而代之的是，可以通过硬件输入来配置以匹配所连接的存储系统。如应用 ARM 9 的目标系统，若要实现支持小端存储系统，则引脚 BIGEND 需要接低电平；若要实现支持大端存储系统，则引脚 BIGEND 需要接高电平。

2.3.6 I/O 端口的访问方式

任务：了解 ARM 体系中 I/O 端口的访问方式。

对于 I/O 端口的访问，ARM 系统采用的标准方法是存储器映射方式（也称为统一编址方式）。该方式为每个 I/O 端口分配特定的存储器地址，当从这些地址读出或写入时，实际完成的是 I/O 功能，即对存储器映射的 I/O 地址进行读取操作时即是输入，而向存储器映射的 I/O 地址进行写入操作时即是输出。

存储器映射的 I/O 端口的读/写操作指令与存储单元的读/写操作指令是相同的，但行为结果通常不同。例如，若对一个存储器单元进行连续两次读取操作，两次读取的数据应该是一样的，除非在两次读取操作间插入了一个对存储单元进行写入的操作。但对存储器映射的 I/O 端口进行连续两次的读取，其值可能不同。这些行为的差异主要会影响到存储系

统中高速缓存和写缓存的使用。也就是说,通常将存储器映射的 I/O 端口标识为非高速缓存的和非缓冲的,以避免改变其访问模式数目、类型、顺序或时序。

2.3.7 异常

任务: 掌握异常的定义,掌握各种异常类型的含义,掌握异常向量的定义,掌握当产生异常时 ARM 微处理器对异常中断的响应过程。

正常的程序执行流程被暂时中断而引发的过程称为异常,例如外部中断信号会引起一个异常的产生。在处理异常之前,当前处理器的状态必须保留,这样当异常处理完成之后,当前程序可以继续执行。处理器允许多个异常同时发生,它们将会按固定的优先级进行处理。

ARM 体系结构中的异常与 8 位/16 位体系结构的中断很相似,但异常与中断的概念并不完全等同。

1. 异常类型

ARM 体系结构所支持的异常类型及其具体含义如表 2-3 所示。

表 2-3 ARM 体系结构支持的异常类型及其具体含义

异常类型	具体含义
复位	当处理器的复位电平有效时,产生复位异常,程序跳转到复位异常处理程序处执行
未定义指令	当 ARM 处理器或协处理器遇到不能处理的指令时,产生未定义指令异常。可使用该异常机制进行软件仿真
软件中断	该异常由执行 SWI 指令产生,可用于用户模式下的程序调用特权操作指令。可使用该异常机制实现系统功能调用
指令预取中止	若处理器预取指令的地址不存在,或该地址不允许当前指令访问,存储器会向处理器发出中止信号,但当预取的指令被执行时,才会产生指令预取中止异常
数据中止	若处理器数据访问指令的地址不存在,或该地址不允许当前指令访问,产生数据中止异常
IRQ(外部中断请求)	当处理器的外部中断请求引脚有效,且 CPSR 中的 I 位为 0 时,产生 IRQ 异常。系统的外围设备可通过该异常请求中断服务
FIQ(快速中断请求)	当处理器的快速中断请求引脚有效,且 CPSR 中的 F 位为 0 时,产生 FIQ 异常

2. 异常向量

异常出现后,强制从异常类型对应的固定存储器地址开始执行程序,这些固定的地址称为"异常向量"。各个异常对应的异常向量如表 2-4 所示。

表 2-4 异常向量

异常类型	模式	正常向量	高向量地址
复位	管理	0x00000000	0xFFFF0000
未定义指令	未定义	0x00000004	0xFFFF0004
软件中断	管理	0x00000008	0xFFFF0008
指令预取中止	中止	0x0000000C	0xFFFF000C
数据中止	中止	0x00000010	0xFFFF0010
IRQ	IRQ	0x00000018	0xFFFF0018
FIQ	FIQ	0x0000001C	0xFFFF001C

3. 对异常的响应

当一个异常出现以后,ARM 微处理器对异常中断的响应过程如下。

(1) 保存处理器当前状态、中断屏蔽位以及各条件标志位,这是通过将当前程序状态寄存器 CPSR 中的内容保存到将要执行的异常中断对应的 SPSR 寄存器中实现的。各异常有自己的物理 SPSR 寄存器。

(2) 设置当前程序状态寄存器 CPSR 中相应的位。包括设置 CPSR 中的位,使用处理器进入相应的执行模式;设置 CPSR 中的位,禁止 IRQ 中断,当进入 FIQ 模式时,禁止新FIQ 中断。

(3) 将寄存器 LR_mode 设置成返回地址。

(4) 将程序计数器(PC)值设置成该异常中断向量地址,从而跳转到相应异常中断处理程序执行。

上述的处理器对异常中断的响应过程可以用如下的伪代码描述:

```
R14_<exception_mode>=return link
SPSR_<exception_mode>=CPSR
CPSR[4:0]=exception mode number
CPSR[5]=0                                  /* 当运行在 AMR 状态下时 */
If <exception_mode>==reset or FIQ then
                                           /* 当响应 FIQ 异常中断时,禁止新的 FIQ 中断 */
CPSR[6]=1                                  /* 禁止新的 FIQ 中断 */
CPSR[7]=1                                  /* 禁止 IRQ 中断 */
PC=exception vector address
```

4. 从异常返回

异常处理完毕之后,ARM 微处理器会执行以下几步操作从异常返回。

(1) 将连接寄存器 LR 的值减去相应的偏移量后送到 PC 中。

(2) 将 SPSR 复制回 CPSR 中。

(3) 若在进入异常处理时设置了中断禁止位,要在此清除。

可以认为应用程序总是从复位异常处理程序开始执行的,因此复位异常处理程序不需要返回。

2.4 ARM微处理器的选型

问题: 当选择 ARM 微处理器时,应从哪几个方面进行考查?

重点: 选择 ARM 微处理器时,应考查几个方面?

内容: ARM 微处理器的选型,包括 ARM 内核的选择、系统的工作频率、片内存储器的容量、片内外围电路的选择四个方面。

由于 ARM 芯片有多达十几种的内核结构,70 多个芯片生产厂家,以及千变万化的内部功能配置组合,给开发人员选择方案带来一定的困难。所以,对 ARM 芯片进行对比研究是十分必要的。

以下针对实际开发设计,就如何选择 ARM 微处理器及所应考虑的主要问题做一些简要的探讨。

选择适合自己需要的 ARM 微处理器主要从几个方面进行考查：内核、系统的工作频率、片内存储器的容量、片内外围电路。

1. ARM 内核的选择

如果希望使用 Windows CE 或 Linux 等操作系统以减少软件开发时间，就需要选择 ARM720T 以上带有 MMU（Memory Management Unit）功能的 ARM 芯片，ARM720T、StrongARM、ARM920T、ARM922T、ARM946T 都带有 MMU 功能。而 ARM7TDMI 没有 MMU，不支持 Windows CE 和大部分的 Linux，但目前有 uCLinux 等少数几种 Linux 不需要 MMU 的支持。

2. 系统的工作频率

系统时钟决定了 ARM 芯片的处理速度。ARM 7 的处理速度为 0.9MIps/MHz，常见的 ARM 7 芯片系统主时钟为 20～133MHz，ARM 9 的处理速度为 1.1MIps/MHz，ARM 9 的系统主时钟通常为 100～233MHz，ARM 10 最高可以达到 700MHz。不同芯片对时钟的处理不同，有的芯片只有一个主时钟频率，这样的芯片可能不能兼顾 UART 和音频时钟准确性，如 Cirrus Logic 的 EP7312 等；有的芯片内部时钟控制器可以分别为 CPU 核和 USB、UART、DSP、音频等功能部件提供同频率的时钟，如 Philips 公司的 SAA7750 等芯片。

3. 片内存储器的容量

如果实际系统中不需要大容量存储器，可以考虑选用有内置存储器的 ARM 芯片，如表 2-5 所示。

表 2-5 内置存储器的 ARM 芯片

芯片型号	供应商	Flash 容量	ROM 容量	SRAM 容量
AT91F40162	Atmel	2MB	256KB	4KB
AT91FR4081	Atmel	1MB		128KB
SAA7750	Philips	384KB		64KB
PUC3030A	Micronas	256KB		56KB
HMS30C7202	Hynix	192KB		
ML67Q4001	OKI	256KB		
LC67F500	Snayo	640KB		32KB

4. 片内外围电路的选择

除 ARM 微处理器核以外，几乎所有的 ARM 芯片均根据各自不同的应用领域扩展了相关功能模块，并集成在芯片之中，称为片内外围电路，如 USB 接口、IIS 接口、LCD 控制器、键盘接口、RTC、ADC 和 DAC、DSP 协处理器等，设计者应分析系统的需求，尽可能采用片内外围电路完成所需的功能，这样既可简化系统的设计，同时提高系统的可靠性。

小结

本章对 ARM 微处理器、ARM 技术的基本概念做了一些简单的介绍，同时也对 ARM 微处理器的体系结构、寄存器的组织、处理器的工作状态、运行模式、处理器异常以及存储模式等内容进行了描述，并就如何对 ARM 微处理器的选型进行了阐述。这些内容是 ARM 体系结构的基本内容，也是嵌入式应用系统软、硬件设计的基础。

第3章

ARM程序设计基础

学习目标

通过本章的学习,应该掌握:

- ARM 指令系统
- 用 ARM 汇编语言编写程序
- ARM 汇编语言与 C/C++ 语言的混合编程

3.1 ARM指令系统

问题：ARM 指令分为哪几类？指令条件码是什么，主要有哪些？ARM 寻址方式有哪些？Thumb 指令有哪些？
重点：ARM 指令及其寻址方式。
内容：ARM 指令系统，ARM 寻址方式，Thumb 指令集。

ARM 微处理器支持 32 位的 ARM 指令集和 16 位的 Thumb 指令集，程序的启动都从 ARM 指令开始，所有的中断或异常都自动转化为 ARM 状态执行。所有 ARM 指令都可以有条件地执行。16 位 Thumb 指令集是从 32 位 ARM 指令集中提取的指令格式，每条 Thumb 指令都有相同功能的 32 位 ARM 指令与之对应。在嵌入式系统开发中，或多或少地都会用到汇编指令，虽然汇编指令可读性差，但是执行速度快，在嵌入式系统的启动过程中常用来初始化与硬件相关的操作。

3.1.1 ARM 指令系统概述

任务：了解 ARM 指令分类，理解 ARM 指令格式和指令条件码。

ARM 处理器的指令集是加载/存储型的，即指令集仅能处理寄存器中的数据，而且处理结果都要放回寄存器中，而对存储器的访问则需要使用专门的加载/存储指令来完成。

1. 指令分类

ARM 微处理器的指令集可以分为跳转指令、数据处理指令、程序状态寄存器处理指令、LOAD/STORE 指令、协处理器指令和异常产生指令 6 大类。基本的指令如表 3-1 所示。

表 3-1　ARM 指令

助记符	指令功能描述	助记符	指令功能描述
ADC	带进位加法指令	LDR	存储器到寄存器的数据传送指令
ADD	加法指令	MCR	从 ARM 寄存器传送到协处理器寄存器
AND	逻辑与指令	MLA	乘加运算指令
B	跳转指令	MOV	数据传送指令
BIC	位清零指令	MRC	从协处理器寄存器到 ARM 寄存器
BL	带返回的跳转指令	MRS	传送 CPRS 到通用寄存器
BLX	带返回和状态切换的跳转指令	MSR	传送通用寄存器到 CPRS
BX	带状态切换的跳转指令	MUL	32 位乘法
CDP	协处理器数据操作指令	MLA	32 位加法
CMN	比较反值指令	MVN	数据取反传送指令
CMP	比较指令	ORR	逻辑或指令
EOR	异或指令	RSB	逆向减法指令
LDC	存储器到协处理器的数据传送指令	RSC	带借位的逆向减法指令
LDM	加载多个寄存器指令	SBC	带借位的减法指令

续表

助记符	指令功能描述	助记符	指令功能描述
STC	协处理器寄存器写入存储器指令	SWI	软件中断指令
STM	批量内存字写入指令	SWP	交换指令
STR	寄存器到存储器的数据传送指令	TEQ	相等测试指令
SUB	减法指令	TST	位测试指令

2. ARM 指令格式

ARM 指令采用固定的 32 位二进制编码，其基本格式如下：

<操作码>{<条件>}{S} <目标寄存器>,<操作数 1 寄存器>{,<操作数 2>}

说明：格式中的＜ ＞表示必选项，{＜ ＞}表示可选项。

其中，

(1) 操作码：即指令助记符，如 ADD 表示加法指令，CMP 表示比较指令。

(2) 条件：表示可选的指令条件码，定义执行条件，当指令没有条件码时为无条件执行。关于条件码的说明见后续说明。

(3) S：为可选后缀，表示该指令的操作结果对 CPSR 的影响。若指定了 S，则根据指令操作结果更新 CPSR 的条件标志位(N、Z、C、V)。

(4) 目标寄存器：为操作结果寄存器。

(5) 操作数 1 寄存器：为存放第一个操作数的寄存器。

(6) 操作数 2：为第 2 个操作数，可以是寄存器，也可以是立即数。

3. 指令条件码

在 ARM 指令 32 位编码中，最高 4 位[31:28]为指令条件码(即 cond)。每种条件码用两个英文缩写字符表示，可添加在指令助记符的后面，表示指令执行时必须要满足的条件。ARM 指令根据 CPSR 中的条件标志位自动判断是否执行该指令。当条件满足时指令执行，否则指令被忽略，继续执行下一条指令。

例如，加法指令 ADD 加上条件后缀 EQ 后成为 ADDEQ，表示“相等则执行加法”，“不相等则本条指令不执行”，即只有当 CPSR 中的 Z 标志为 1 时，才会执行该加法指令。

在 4 位条件码形成的 16 位条件码中只有 15 种可供用户使用，而 1111 为系统保留，如表 3-2 所示。

表 3-2 条件码的含义

条件码[31:28]	助记符后缀	解　　释	CPSR 标志位状态
0000	EQ	相等	Z 置位
0001	NE	不相等	Z 清零
0010	CS/HS	进位/无符号数大于或等于	C 置位
0011	CC/LO	无进位/无符号数小于	C 清零
0100	MI	负数	N 置位
0101	PL	正数或零	N 清零

续表

条件码[31:28]	助记符后缀	解　　释	CPSR 标志位状态
0110	VS	溢出	V 置位
0111	VC	未溢出	V 清零
1000	HI	无符号数大于	C 置位且 Z 清零
1001	LS	无符号数小于或等于	C 清零且 Z 置位
1010	GE	有符号数大于或等于	N 等于 V
1011	LT	有符号数小于	N 不等于 V
1100	GT	有符号数大于	Z 清零且 N 等于 V
1101	LE	有符号数小于或等于	Z 置位且 N 不等于 V
1110	AL	总是	任何状态

当指令中无条件码时，默认为 AL，即该指令总是执行（无条件执行）。

3.1.2　ARM 寻址方式

任务：掌握 ARM 指令的寻址方式分类，理解每一种寻址方式的含义。

所谓寻址方式就是处理器根据指令中给出的地址信息来确定物理地址的方式。目前 ARM 指令系统支持的基本寻址方式有 7 种，分别介绍如下。

1. 立即寻址

立即寻址也称为立即数寻址，这种寻址方式就是在指令中给出操作数，只要取出指令也就取到了操作数，这个操作数称为立即数，对应的寻址方式叫做立即寻址。

例如：

```
ADD R0,R0,#1              ;R0=R0+1,1 为立即寻址
AND R2,R1,#0xFF           ;R0=R0 AND #0xFF,0xFF 为立即寻址
```

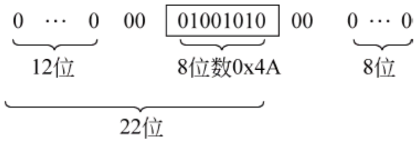
第一条指令完成寄存器 R0 的内容加 1，结果送回 R0 中。第 2 条指令完成 R1 的值与 #0xFF 的“与”运算，结果送到 R2 中。

在上述指令中，操作数 2 即为立即数，要求以 # 为前缀，在 # 后加 0x 或 & 表示十六进制数，在 # 后加 0b 表示二进制数，在 # 后加 0d 或省略表示十进制数。

思考：一条 ARM 指令有 32 位编码，如果指令中出现一个 32 位的立即数，该如何表示？

在 ARM 指令编码中，32 位的有效立即数是用 12 位编码间接表示的，12 位编码分成两部分，前 4 位表示移位位数，后 8 位表示一个常数，32 位的立即数由 8 位常数循环右移 2× 移位位数得到。例如 0x0000F200 可以由 8 位的 0xF2 循环右移 24(2×12)位得到，因此，立即数 0x0000F200 在 ARM 指令中的编码表示为 0xCF2(C 为 12 的十六进制表示形式)。

再如立即数 0x00012800，其二进制形式为：



因此，该立即数可以由 0x4A 循环右移 22(2×11)位得到，因此在 ARM 指令中该立即数的编码为 0xB4A。

但并不是所有的常数都是合法的立即数，例如：0x1010、0x102、0xF1F 则不能通过上述的合法方法表示。读者可自行验证。

2. 寄存器寻址

寄存器寻址就是将寄存器中的数值作为操作数，指令中的地址码给出的是寄存器的编号。例如：

```
ADD R0,R1,R2                ;R0←R1+R2
```

该指令的含义是将 R1 和 R2 的内容相加，结果放在 R0 中。

在寄存器寻址中有一种特殊的使用方式，即第二操作数在和第一个操作数结合之前先进行移位操作，这种方式也称为寄存器移位寻址。移位位数可以是 5 位立即数或寄存器。例如：

```
ADD R0,R1,R2,LSL #3
```

该指令的含义是 R2 的内容先逻辑左移 3 位，再与寄存器 R1 的内容相加，结果放入 R0 中。需要注意的是，在移位操作中，第二操作数必须是寄存器，而且指令执行完毕后第二操作数寄存器的内容不变，参与运算的是第二操作数移位的中间结果，但这个中间结果并不保存。

可进行的移位操作有以下几种。

(1) LSL：逻辑左移(logical shift left)，空出的最低有效位用 0 填充。

(2) LSR：逻辑右移(logical shift right)，空出的最高有效位用 0 填充。

(3) ASL：算术左移(arithmetic shift left)，空出的最低有效位用 0 填充，因此它与 LSL 的用法相同。

(4) ASR：算术右移(arithmetic shift right)，算术移位的对象是有符号数，在移位过程中必须保持操作数的符号不变，即如果源操作数是正数，空出的最高有效位用 0 填充，如果是负数，则用 1 填充。

(5) ROR：循环右移(rotate right)，移出的字的最低有效位依次填入空出的最高有效位。

(6) RRX：带扩展的循环右移(rotate right with extend)，将寄存器的内容循环右移 1 位，空位用原来的 C 标志位填充。当移位类型为 RRX 时，无须指定移位类型。

3. 寄存器间接寻址

寄存器间接寻址就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中，寄存器起到地址指针的作用。例如：

```
LDR R0,[R1]                ;R0←[R1]
STR R0,[R1]                ;R0→[R1]
```

第一条指令是将寄存器 R1 所指向的地址单元的内容装载到寄存器 R0 中，第二条指令是将寄存器 R0 的内容存储到 R1 寄存器所指向的地址单元。

4. 基址变址寻址

基址变址寻址就是将基址寄存器的内容与指令中给出的地址偏移量相加，得到一个操

作数的有效地址，用于访问基址附近的存储单元。

采用基址变址寻址方式的指令通常有以下几种形式：

```
LDR R0,[R1,#4]          ;R0←[R1+4]
LDR R0,[R1],#4          ;R0←[R1]、R1=R1+4
LDR R0,[R1,#4]!         ;R0←[R1+4]、R1=R1+4
LDR R0,[R1,R2]          ;R0←[R1+R2]
```

第一条指令是将寄存器 R1 的内容加上 4 得到操作数的有效地址，取出操作数并存入 R0 中。

第二条指令是将 R1 的内容作为操作数的有效地址，取得操作数并存入寄存器 R0 中，然后 R1 的内容自增 4 个字节，即 R1 指向下一个字的有效地址。

第三条指令是将 R1 的内容加上 4 作为操作数的有效地址，取出操作数并存入寄存器 R0 中，然后 R1 的内容自增 4 个字节。

第四条指令是将 R1 的内容加上 R2 的内容作为操作数的有效地址，取出操作数并存入寄存器 R0 中。

5. 多寄存器寻址

多寄存器寻址是指一条指令可以一次传递多个寄存器的值。这种寻址方式允许一条指令一次最多传送 16 个通用寄存器的值。例如：

```
LDMIA R0,{R1,R2,R4}          ;R1←[R0]、R2←[R0+4]、R4←[R0+8]
```

这条指令是将 R0 指向的连续的存储空间的内容分别送入 R1、R2、R4 中。

6. 相对寻址

相对寻址可以看做是将程序计数器 PC 作为基址的一种基址变址寻址方式。指令的地址标号作为位移量，与 PC 相加得到操作数的有效地址。例如在下面的程序段中 BL 指令采用了相对寻址方式。

```
    BL SUBR             ;调用子程序 SUBR
    …                   ;返回位置
SUBR …                  ;子程序入口地址
    MOV PC,R14          ;返回
```

7. 堆栈寻址

堆栈是一种按照“先进后出”或“后进先出”方式进行数据存取的存储区。指向堆栈的地址寄存器称为堆栈指针(SP)，堆栈的访问是通过将堆栈指针(R13，ARM 处理器的不同工作模式对应的物理寄存器各不相同)指向一块存储区域(堆栈)来实现的。

堆栈根据其内存地址增长的方向分为递增堆栈和递减堆栈。递增堆栈指访问存储器时，存储器地址由低地址向高地址方向增长；递减堆栈指访问存储器时，地址由高地址向低地址方向增长。

根据堆栈指针指向数据位置的不同，又可分为满堆栈和空堆栈。若堆栈指针指向最后压入堆栈的数据，则该栈称为满堆栈；若堆栈指针指向下一个即将压入的数据将要放入的空位置，则该栈称为空堆栈。

递增、递减、空堆栈、满堆栈进行组合可以产生 4 种类型的堆栈。

(1) 满递增堆栈：堆栈指针指向最后压入的数据，而且压入数据时堆栈由低地址向高地址生成。

(2) 空递增堆栈：堆栈指针指向下一个要压入的数据的空位置，而且压入数据时堆栈由低地址向高地址生成。

(3) 满递减堆栈：堆栈指针指向最后压入的数据，而且压入数据时堆栈由高地址向低地址生成。

(4) 空递减堆栈：堆栈指针指向下一个要压入的数据的空位置，而且压入数据时堆栈由高地址向低地址生成。

在 ARM 指令中，堆栈寻址通过 LOAD/STORE 指令来实现，在 Thumb 指令中，堆栈寻址通过 PUSH/POP 指令来实现。

3.1.3 ARM 指令集

任务：理解每一种指令的含义。

1. 数据处理指令

ARM 数据处理指令包括算术运算指令、逻辑运算指令、数据传送指令、比较指令、测试指令、乘法指令。

(1) 算术运算指令——ADD、SUB、RSB、ADC、SBC、RSC

指令格式：操作码{条件}{S}目标寄存器，操作数 1 寄存器，操作数 2

指令功能：用于加、减、反减等算术运算，包括带进位的算术运算。

① ADD 指令用于将操作数 1 寄存器的值和操作数 2 相加。

② SUB 指令用于将操作数 1 寄存器的值减去操作数 2。

③ RSB 指令用于将操作数 2 的值减去操作数 1。反减的优点在于操作数 2 的可选范围比较大。

④ ADC、SBC、RSC 指令分别是 ADD、SUB、RSB 的带进(借)位的运算，运算结果将影响 CPSR 中的进位标志 C。

范例 1：完成 64 位整数相加。

```
ADDS  R4,R0,R2        ;加低位有效字
ADC   R5,R1,R3        ;加高位有效字（连同低位进位）
```

范例 2：完成 96 位减法。

```
SUBS  R3,R6,R9
SBCS  R4,R7,R10
SBC   R5,R8,R11
```

(2) 逻辑运算指令——AND、ORR、EOR、BIC

指令格式：操作码{条件}{S}目标寄存器，操作数 1 寄存器，操作数 2

指令功能：AND、ORR、EOR 分别完成逻辑与、逻辑或、逻辑异或运算，BIC 指令用于将操作数 1 寄存器中的位与操作数 2 中相应位的反码进行“与”运算，该指令可以将操作数

1 的某些位清 0。

范例：

```
AND  R0,R0,#00FF          ;将 R0 的高 24 位清 0,低 8 位保持不变
ORR  R0,R0,#00FF          ;将 R0 的低 8 位置 1,高 24 位保持不变
EOR  R0,R0,#00FF          ;将 R0 的低 8 位反转,高 24 位保持不变
```

(3) 数据传送指令——MOV 和 MVN

指令格式：操作码{条件}{S}目标寄存器，源操作数

指令功能：

① MOV 指令用于将源操作数的值送往目标寄存器。

② MVN 是“取反传送”指令，用于将源操作数按位取反后的结果送往目标寄存器。

范例：

```
MOV R0,R1                 ;将 R1 的值传送到 R0
MOV PC,R14                ;将 R14 的值传送到 PC,常用于从子程序返回
MOVS R1,R0,LSL #2         ;将 R0 的值左移 2 位后传送到 R1
MVN R1,R0                 ;将 R0 的值按位取反后传送到 R1
```

(4) 比较指令——CMP 和 CMN

指令格式：操作码{条件} 操作数 1 寄存器，操作数 2

指令功能：

① CMP 指令用于把操作数 1 寄存器的值与操作数 2(寄存器或立即数)的值进行比较，同时更新 CPSR 中条件标志位的值。该操作实际上进行了一次减法运算，但不保存结果，只改变条件标志位。

② CMN 是“取反比较”指令，用于将操作数 1 和操作数 2 相加，并根据结果修改条件标志位。

范例：

```
CMP R1,R0
CMN R1,#50
```

(5) 测试指令——TST 和 TEQ

指令格式：操作码{条件}操作数 1 寄存器，操作数 2

指令功能：

① TST 是位测试指令，用于对两个操作数进行按位“与”操作，并根据结果更新条件标志位，通常用于测试寄存器中的某些位是 1 还是 0。

② TEQ 是相等测试指令，用于对两个操作数进行按位“异或”运算，并根据结果更新条件标志位，通常用于比较两个操作数是否相等。

范例：

```
TSTNE          R0,#0x3
TEQEQ          R10,R9
```

(6) 乘法指令

乘法指令完成两个 32 位寄存器数据的乘法运算，运算结果分为 32 位和 64 位数据。乘

法指令一共有 6 种格式，如表 3-3 所示。

表 3-3 乘法指令的 6 种格式

助记符	说明	操作
MUL Rd，Rm，Rs	32 位乘法指令	Rd←Rm×Rs(Rd≠Rm)
MLA Rd，Rm，Rs，Rn	32 位乘加指令	Rd←Rm×Rs＋Rn(Rd≠Rm)
UMULL RdLo，RdHi，Rm，Rs	64 位无符号乘法指令	(RdLo，RdHi)←Rm×Rs
UMLAL RdLo，RdHi，Rm，Rs	64 位无符号乘加指令	(RdLo，RdHi)←Rm×Rs＋(RdLo，RdHi)
SMULL RdLo，RdHi，Rm，Rs	64 位带符号乘法指令	(RdLo，RdHi)←Rm×Rs
SMLAL RdLo，RdHi，Rm，Rs	64 位带符号乘加指令	(RdLo，RdHi)←Rm×Rs＋(RdLo，RdHi)

① 32 位乘法指令——MUL。

指令格式：MUL{条件}{S}目的寄存器，源操作数 1 寄存器，源操作数 2 寄存器

指令功能：将源操作数 1 寄存器和源操作数 2 寄存器中的值相乘，将结果的低 32 位保存到目的寄存器中。特别说明的是，目的寄存器和源操作数 1 寄存器不能是同一个寄存器。

范例：

```
MUL R1,R2,R3            ;R1=R2×R3
MULS R0,R2,R5           ;R0=R2×R5,并根据结果设置 CPSR 中的 Z 位和 N 位
```

② 32 位乘加指令——MLA。

指令格式：MLA{条件}{S}目的寄存器，源操作数 1 寄存器，源操作数 2 寄存器，加数寄存器

指令功能：

将源操作数 1 寄存器和源操作数 2 寄存器中的值相乘，再加上加数寄存器的数据，将结果的低 32 位保存到目的寄存器中。特别说明的是，目的寄存器和源操作数 1 寄存器不能是同一个寄存器。

范例：

```
MLA R1,R2,R3,R0          ;R1=R2×R3+R0
```

③ 64 位无符号乘法指令——UMULL。

指令格式：UMULL{条件}{S}目的寄存器 Lo，目的寄存器 Hi，源操作数 1 寄存器，源操作数 2 寄存器

指令功能：将源操作数 1 寄存器和源操作数 2 寄存器中的值作为无符号数相乘，将结果的低 32 位保存到目的寄存器 Lo 中，高 32 位保存到目的寄存器 Hi 中。

范例：

```
UMULL R0,R1,R4,R6          ;(R1,R0)=R4×R6
```

④ 64 位无符号乘加指令——UMLAL。

指令格式：UMLAL{条件}{S}目的寄存器 Lo，目的寄存器 Hi，源操作数 1 寄存器，源操作数 2 寄存器

指令功能：将源操作数 1 寄存器和源操作数 2 寄存器中的值作为无符号数相乘，将乘

积结果与目的寄存器 Hi,目的寄存器 Lo 相加,结果的低 32 位保存到目的寄存器 Lo 中,高 32 位保存到目的寄存器 Hi 中。

范例:

```
UMLAL R0,R1,R4,R6              ;(R1,R0)=R4×R6+(R1,R0)
```

⑤ 64 位带符号乘法指令——SMULL。

指令格式:SMULL{条件}{S}目的寄存器 Lo,目的寄存器 Hi,源操作数 1 寄存器,源操作数 2 寄存器

指令功能:将源操作数 1 寄存器和源操作数 2 寄存器中的值作为带符号数相乘,结果的低 32 位保存到目的寄存器 Lo 中,高 32 位保存到目的寄存器 Hi 中。

范例:

```
SMULL R2,R3,R6,R7              ;(R3,R2)=R6×R7
```

⑥ 64 位带符号乘加指令——SMLAL。

指令格式:SMLAL{条件}{S}目的寄存器 Lo,目的寄存器 Hi,源操作数 1 寄存器,源操作数 2 寄存器

指令功能:将源操作数 1 寄存器和源操作数 2 寄存器中的值作为带符号数相乘,将乘积结果与(目的寄存器 Hi,目的寄存器 Lo)相加,结果的低 32 位保存到目的寄存器 Lo 中,高 32 位保存到目的寄存器 Hi 中。

范例:

```
SMLAL R0,R1,R5,R8              ;(R1,R0)=R5×R8+(R1,R0)
```

2. 跳转指令

跳转指令用于实现程序流程的转移,在 ARM 中可以采用两种方法实现程序流程的转移。一种方法是直接向 PC 寄存器(R15)中写入转移的目标地址值,通过改变 PC 的值实现程序的跳转;另一种方法是使用跳转指令。

使用 ARM 的跳转指令可以从当前指令向前或向后的 32MB 空间跳转,包括 B(简单跳转指令)、BL(带返回的跳转指令)、BX(带状态切换的跳转指令)、BLX(带返回和状态切换的跳转指令)4 条指令。

(1) 简单跳转指令 B

指令格式:B{条件}目标地址

指令功能:跳转到目标地址处执行。

范例:

```
B LABEL                        ;程序无条件跳转到标号 LABEL 处执行
B 0x1400                       ;跳转到绝对地址 0x1400 处执行
```

利用 B 指令实现循环:

```
    MOV R0,#10                 ;初始化循环计数器
LOOP …
    SUBS R0,#1                 ;计数器减 1,并设置条件码
    BNE LOOP                   ;如果计数器 R0 不为 0,则继续循环
```

```
    …                            ;否则终止循环
```

(2) 带返回的跳转指令 BL

指令格式：BL{条件}　　目标地址

指令功能：在跳转之前将 PC 的当前值保存到 R14 中，因此可以通过将 R14 的内容重新加载到 PC 中来返回到跳转指令之后的那个指令处执行。该指令是实现自程序调用的基本手段。

范例：

```
    …
    BL SUBP                     ;子程序调用 (PC→R14)
    …                           ;返回到这里
    …
SUBP …                          ;子程序入口
    …
    MOV PC,R14                  ;子程序返回
```

(3) 带状态切换的跳转指令 BX

指令格式：BX{条件}目标地址寄存器

指令功能：指令执行时将目标地址寄存器的第 0 位复制到 CPSR 的 T 标志位(决定程序是切换到 Thumb 指令还是继续执行 ARM 指令)，[31:1]位移入 PC：

若目标地址第 0 位为 0，则处理器执行 ARM 指令；若目标地址第 0 位为 1，则处理器跳转到 Thumb 指令执行。

范例：

```
BX R0                           ;跳转到 R0 指定地址,并根据 R0 的最低位切换处理器状态
```

(4) 带返回和状态切换的跳转指令 BLX

指令格式：BLX 标号　或　BLX{条件}目标地址寄存器

指令功能：将下一条指令的地址复制到 R14 中，转移到标号处或目标地址寄存器指定的位置；如果目标地址寄存器的第 0 位为 1 或使用标号，则程序切换到 Thumb 状态。

范例：

```
      CODE32                    ;以下是 ARM 代码
      …
      BLX Tsub                  ;调用 Thumb 子程序
      …
      CODE16                    ;开始 Thumb 代码
Tsub  …                         ;Thumb 子程序
      BX R14                    ;返回到 ARM 代码
```

3. 程序状态寄存器处理指令

ARM 微处理器支持程序状态寄存器访问指令，用于在程序状态寄存器和通用寄存器之间传送数据，程序状态寄存器访问指令包括 MRS 和 MSR 两条指令。

(1) MRS 指令

指令格式：MRS{条件}通用寄存器，程序状态寄存器(CPSR 或 SPSR)

指令功能：将程序状态寄存器的内容传送到通用寄存器中。注意通用寄存器不能使用 R15。该指令一般用在以下两种情况下。

① 当需要改变程序状态寄存器的内容时，可用 MRS 将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器。

② 当进行异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态寄存器的值，然后保存。

范例：

```
MRS R0,CPSR                    ;传送 CPSR 的内容到 R0
MRS R1,SPSR                    ;传送 SPSR 的内容到 R1
```

(2) MSR 指令

指令格式：MSR{条件}程序状态寄存器_<域>,操作数

指令功能：将操作数的内容传送到程序状态寄存器(CPSR 或 SPSR)的特定域中。其中，操作数可以为通用寄存器或立即数。<域>用于设置程序状态寄存器中需要操作的位，为可选项，32 位的程序状态寄存器可分为以下 4 个域(必须用小写字母表示)。

① 位[31:24]为条件标志位域，用 f 表示。

② 位[23:16]为状态位域，用 s 表示。

③ 位[15:8]为扩展位域，用 x 表示。

④ 位[7:0]为控制位域，用 c 表示。

该指令通常用于恢复或改变程序状态寄存器的内容，在使用时，一般要在 MSR 指令中指明将要操作的域。

范例 1：

```
MSR CPSR,R0                    ;传送 R0 的内容到 CPSR
MSR SPSR,R0                    ;传送 R0 的内容到 SPSR
MSR CPSR_c,R0                  ;传送 R0 的内容到 CPSR,但仅仅修改 CPSR 中的控制位域
```

由上述介绍可知，使用 MRS 指令读取 CPSR 的值后，可以判断 ALU 的状态标志，或 IRQ、FIQ 中断是否允许等；在异常处理程序中，读 SPSR 可以知道进入异常之前的状态等。MRS 与 MSR 配合使用，可以实现 CPSR、SPSR 寄存器的“读-修改-写”操作，从而实现处理器的模式切换、允许/禁止 IRQ/FIQ 中断设置等。另外，当切换进程或允许异常中断嵌套时，也需要使用 MRS 指令读取 SPSR 的值并保存起来。

范例 2：

使能 IRQ 中断：

```
ENABLE_IRQ
    MRS R0,CPSR
    BIC R0,R0,#0X80
    MSR CPSR_c,R0
    MOV PC,LR
```

禁能 IRQ 中断：

```
DISABLE_IRQ
```

```
MRS R0,CPSR
ORR R0,R0,#0X80
MSR CPSR_c,R0
MOV PC,LR
```

4. 寄存器存取指令

由于 ARM 处理器是基于 RISC 指令系统的，不能直接对内存进行操作，只能存取寄存器中的数据。寄存器存取指令用于完成寄存器和内存之间的数据传递。该指令分为 3 种类型：单寄存器传送指令、多寄存器传送指令、交换指令。

单寄存器传送指令 LDR/STR 的作用是将单一的数据传入（LDR）或传出（STR）寄存器。对内存的访问可以是 32 位的字、16 位有符号/无符号的半字或 8 位有符号/无符号的字节数据。多寄存器传送指令 LDM/STM 可以用一个指令加载/存储多个寄存器的数据，大大提高存取效率。交换指令 SWP 是交换寄存器和存储器内容的指令，可用于信号量操作等。

（1）单寄存器传送指令——LDR/STR

指令格式如下：

```
LDR{条件}        Rd,<地址>          ;把<地址>处的一个字数据装入寄存器 Rd
STR{条件}        Rd,<地址>          ;把寄存器 Rd 的一个字数据保存到<地址>中
LDR{条件}{S}H    Rd,<地址>          ;把<地址>处的半字数据装入寄存器 Rd
STR{条件}{S}H    Rd,<地址>          ;把寄存器 Rd 的半字数据保存到<地址>中
LDR{条件}{S}B    Rd,<地址>          ;把<地址>处的一个字节数据装入寄存器 Rd
STR{条件}{S}B    Rd,<地址>          ;把 Rd 的字节数据 (即低 8 位)保存到<地址>中
```

其中，选项{S}表示传送带符号的数据。

LDR/STR 指令的寻址是非常灵活的，通常由两部分组成，一部分为一个基址寄存器，可以为任何通用寄存器；另一部分是地址偏移量，地址偏移量有以下 3 种形式。

① 立即数。立即数可以是一个无符号数，这个数可加到基址寄存器上，也可用基址寄存器值减去立即数。

范例：

```
LDR   R1,[R0,#0x08]       ;将 R0+0x08 地址处的数据传送给 R1,R0 不变
LDR   R1,[R0,#-0x08]      ;将 R0-0x08 地址处的数据传送给 R1,R0 不变
LDR   R1,[R0]             ;将 R0 地址处的数据传送给 R1(零偏移量)
LDR   R1,[R0,#4]!         ;首先 R0=R0+4,即基址寄存器发生变化,然后将新的 R0 地址处的数据
                          传送给 R1(注意:基址寄存器先改变,该寻址方式又称为前变址寻址)
LDR   R1,[R0],#4          ;首先将 R0 地址处的数据传送给 R1,然后使 R0=R0+4(注意:基址寄存
                          器后改变,该寻址方式又称为后变址寻址)
```

② 寄存器。寄存器中的值可以加到基址寄存器中，也可以用基址寄存器的值减去偏移寄存器的值。

范例：

```
LDR   R1,[R0,R2]          ;将 R0+R2 地址的数据传送到 R1,R0 不变
LDR   R1,[R0,-R2]         ;将 R0-R2 地址的数据传送到 R1,R0 不变
```

③ 寄存器及移位常数。寄存器移位后的值可以加到基址寄存器上，也可以用基址寄存器减去这个值。

范例：

```
LDR  R1,[R0,R2 LSL #2]        ;将 R0+R2×4(即左移两位)地址处的数据传送到 R1
LDR  R1,[R0,-R2 LSL #2]       ;将 R0-R2×4(即左移两位)地址处的数据传送到 R1
```

(2) 多寄存器传送指令——LDM 和 STM

多寄存器传送指令 LDM 和 STM 用于实现多个寄存器和一块连续的内存单元之间的字数据传递。LDM 指令用于加载寄存器，即将连续内存单元的数据加载到多个寄存器中；STM 指令用于存储寄存器，即将多个寄存器的数据存储到连续的内存单元中。指令格式如下：

```
LDM{条件}<模式>Rn{!},reglist{^}
STM{条件}<模式>Rn{!},reglist{^}
```

其中，<模式>有如下 8 种(其中前 4 种用于数据块的传递，后 4 种用于堆栈操作)。

IA：每次传送数据后地址增加 4(指向下一个字数据)。

IB：每次传送数据前地址增加 4。

DA：每次传送数据后地址减小 4(指向前一个字数据)。

DB：每次传送数据前地址减小 4。

FD：满递减堆栈。

ED：空递减堆栈。

FA：满递增堆栈。

EA：空递增堆栈。

其中，Rn 为基址寄存器，装有传送数据的初始地址，Rn 不允许是 R15；后缀{!}表示最后的地址写回到 Rn 中，即改变 Rn 的值。

其中，reglist 寄存器列表表示多个寄存器或寄存器范围，用“，”分隔，例如{R0，R2，R4-R6}，寄存器按编号由小到大排列。后缀{^}不允许在用户模式或系统模式下使用，若在 LDM 指令且寄存器列表中包含 PC(即 R15)时使用，则除了进行正常的多寄存器数据传送外，还同时将 SPSR 数据传给 CPSR，可用于异常处理返回时恢复现场；使用后缀{!}且寄存器列表不包含 PC 时，加载/存储的是用户模式的寄存器，而不是当前模式的寄存器。

范例：

```
LDMIA   R0!,{R1-R5}        ;将 R0 地址处连续的字数据加载到 R1-R5 中，R0 的值改变
STMIA   R1!,{R3-R7}        ;将 R3-R7 中的数据存储到 R1 所指的连续地址中，R1 改变
STMFD   SP!,{R0-R7,LR}     ;现场保护，将{R0-R7,LR}数据入栈
LDMFD   SP!,{R0-R7,PC}     ;将堆栈中保护的各个寄存器数据恢复到{R0-R7,PC}
```

(3) 寄存器/存储器交换指令——SWP

该指令用于将一个内存单元(该内存单元的地址存放在寄存器 Rn 中)的数据读入到一个寄存器 Rd 中，同时将另一个寄存器 Rm 的数据写入到该内存单元中。指令格式如下：

```
SWP{条件}{B}Rd,Rm,[Rn]
```

其中，使用选项 B 表示交换字节数据，否则交换 32 位的字数据。若 Rm 与 Rn 相同，则寄存

器和内存数据进行交换。注意：Rn 和 Rd、Rm 不能相同。

范例：

```
SWP R1,R1,[R0]          ;交换[R0]和 R1 的数据
SWP R1,R2,[R0]          ;将[R0]地址的数据存到 R1,同时将 R2 的数据写入到内存[R0]中
```

5. 协处理器指令

ARM 微处理器可支持多达 16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略 ARM 处理器和其他协处理器的指令。ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作，以及在 ARM 处理器的寄存器和协处理器的寄存器之间传送数据，和在 ARM 协处理器的寄存器和存储器之间传送数据。ARM 协处理器指令包括以下 5 个。

（1）协处理器数操作指令——CDP

CDP 指令用于 ARM 处理器通知 ARM 协处理器执行特定的操作，若协处理器不能成功完成特定的操作，则产生未定义指令异常。

CDP 指令的格式为：

```
CDP{条件} 协处理器编码,协处理器操作码 1,目的寄存器,源寄存器 1,源寄存器 2,协处理器操作码 2
```

其中，协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，目的寄存器和源寄存器均为协处理器的寄存器，指令不涉及 ARM 处理器的寄存器和存储器。

范例：

```
CDP P3,2,C12,C10,C3,4          ;完成协处理器 P3 的初始化
```

（2）协处理器数据加载指令——LDC

LDC 指令用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。

LDC 指令的格式为：

```
LDC{条件}{L}协处理器编码,目的寄存器,[源寄存器]
```

其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。

范例：

```
LDC P3,C4,[R0]
        ;将 ARM 处理器的寄存器 R0 所指向的存储器中的字数据传送到协处理器 P3 的寄存器 C4 中
```

（3）协处理器数据存储指令——STC

STC 指令用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。

STC 指令的格式为：

```
STC{条件}{L}协处理器编码,源寄存器,[目的寄存器]
```

其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。

范例：

```
STC P3,C4,[R0]
            ;将协处理器 P3 的寄存器 C4 中的字数据传送到 ARM 处理器的寄存器 R0 所指向的存储器中。
```

(4) ARM 处理器寄存器到协处理器寄存器的数据传送指令——MCR

MCR 指令用于将 ARM 处理器寄存器中的数据传送到协处理器寄存器中，若协处理器不能成功完成操作，则产生未定义指令异常。

MCR 指令的格式为：

```
MCR{条件}协处理器编码,协处理器操作码 1,源寄存器,目的寄存器 1,目的寄存器 2,协处理器操作码 2
```

其中，协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，源寄存器为 ARM 处理器的寄存器，目的寄存器 1 和目的寄存器 2 均为协处理器的寄存器。

范例：

```
MCR P3,3,R0,C4,C5,6
                    ;将 ARM 处理器寄存器 R0 中的数据传送到协处理器 P3 的寄存器 C4 和 C5 中
```

(5) 协处理器寄存器到 ARM 处理器寄存器的数据传送指令——MRC

MRC 指令用于将协处理器寄存器中的数据传送到 ARM 处理器寄存器中，若协处理器不能成功完成操作，则产生未定义指令异常。

MRC 指令的格式为：

```
MRC{条件}协处理器编码,协处理器操作码 1,目的寄存器,源寄存器 1,源寄存器 2,协处理器操作码 2
```

其中，协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，目的寄存器为 ARM 处理器的寄存器，源寄存器 1 和源寄存器 2 均为协处理器的寄存器。

范例：

```
MRC P3,3,R0,C4,C5,6       ;将协处理器 P3 的寄存器 C4、C5 中的数据传送到 ARM 处理器的寄存器 R0 中
```

关于具体型号的 ARM 处理器的相关协处理器及其寄存器配置、操作码等内容可参考 ARM 处理器说明书。

6. 异常产生指令

ARM 微处理器所支持的异常指令有两条：软件中断指令 SWI 和断点中断指令 BKPT。

(1) 软件中断指令——SWI

SWI 指令用于产生软件中断，以便用户程序能调用操作系统的系统例程。操作系统在 SWI 的异常处理程序中提供相应的系统服务。

SWI 指令的格式为：

```
SWI{条件}24 位的立即数
```

其中，24 位的立即数用于指定用户程序调用系统例程的类型，相关参数通过通用寄存器传递，当指令中 24 位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器R0 的内容决定，同时，参数通过其他通用寄存器传递。

范例：

```
SWI 0x02                  ;调用操作系统编号为 02 的系统例程
```

（2）断点中断指令——BKPT

BKPT 指令用于产生软件断点中断，以便进行程序的调试。

BKPT 指令的格式为：

```
BKPT 16位的立即数
```

3.1.4 Thumb 指令集

任务：了解 Thumb 指令集与 ARM 指令集的区别，了解 Thumb 状态寄存器与 ARM 状态寄存器的关系。

为兼容数据总线宽度为 16 位的应用系统，ARM 体系结构除了支持执行效率很高的 32 位ARM 指令集以外，还支持 16 位的 Thumb 指令集。Thumb 指令集是 ARM 指令集的一个子集，是针对代码密度问题而提出的，它具有 16 位的代码宽度。与等价的 32 位代码相比较，Thumb 指令集在保留 32 位代码优势的同时，极大地节省了系统的存储空间。Thumb 不是一个完整的体系结构，处理器不会只执行 Thumb 指令集而不支持 ARM 指令集。

当处理器在执行 ARM 程序段时，称 ARM 处理器处于 ARM 工作状态，当处理器在执行 Thumb 程序段时，称 ARM 处理器处于 Thumb 工作状态。Thumb 指令集并没有改变 ARM 体系底层的编程模型，只是在该模型上增加了一些限制条件，只要遵循一定的调用规则，Thumb 子程序和 ARM 子程序就可以互相调用。

与 ARM 指令集相比较，Thumb 指令集中的数据处理指令的操作数仍然是 32 位，指令地址也为 32 位，但 Thumb 指令集为实现 16 位的指令长度，舍弃了 ARM 指令集的一些特性，如大多数的 Thumb 指令是无条件执行的，而几乎所有的 ARM 指令都是有条件执行的，大多数的 Thumb 数据处理指令采用 2 地址格式。由于 Thumb 指令的长度为 16 位，即只用 ARM 指令一半的位数来实现同样的功能，所以，要实现特定的程序功能，所需的 Thumb 指令的条数较 ARM 指令多。在一般情况下，Thumb 指令与 ARM 指令的时间效率和空间效率关系如下。

（1）Thumb 代码所需的存储空间约为 ARM 代码的 60%～70%。

（2）Thumb 代码使用的指令数比 ARM 代码多 30%～40%。

（3）若使用 32 位的存储器，ARM 代码比 Thumb 代码快约 40%。

（4）若使用 16 位的存储器，Thumb 代码比 ARM 代码快 40%～50%。

（5）与 ARM 代码相比较，使用 Thumb 代码，存储器的功耗会降低约 30%。

显然，ARM 指令集和 Thumb 指令集各有其优点，若对系统的性能有较高要求，应使用 32 位的存储系统和 ARM 指令集，若对系统的成本及功耗有较高要求，则应使用 16 位的存储系统和 Thumb 指令集。当然，若将两者结合起来，充分发挥其各自的优点，会取得更好的效果。

Thumb 指令集与 ARM 指令集在以下几个方面有区别。

（1）跳转指令。条件跳转在范围上有更多的限制，转向子程序只具有无条件转移功能。

（2）数据处理指令。对通用寄存器进行操作，操作结果需放入其中一个操作数寄存器，而不是第三个寄存器。

（3）单寄存器加载和存储指令。在 Thumb 状态下，单寄存器加载和存储指令只能访问

R0～R7。

(4) 批量寄存器加载和存储指令。LDM 和 STM 指令可以加载或存储任何范围为 R0～R7 的寄存器子集,PUSH 和 POP 指令使用堆栈指针 R13 作为基址实现满递减堆栈,除 R0～R7 外,PUSH 指令还可以存储连接寄存器 R14,并且 POP 指令可以加载程序指令 PC。

(5) Thumb 指令集没有包含进行异常处理时需要的一些指令,因此,在产生异常中断时还需要使用 ARM 指令。这种限制决定了 Thumb 指令不能单独使用,而是需要与 ARM 指令配合使用。

Thumb 状态寄存器集是 ARM 状态寄存器集的子集,程序员可直接访问 8 个通用寄存器 R0～R7、PC、堆栈指针 SP、连接寄存器 LR 和 CPSR。每个特权模式都有分组的 SP、LR 和 SPSR。

Thumb 寄存器和 ARM 寄存器之间的关系如图 3-1 所示。

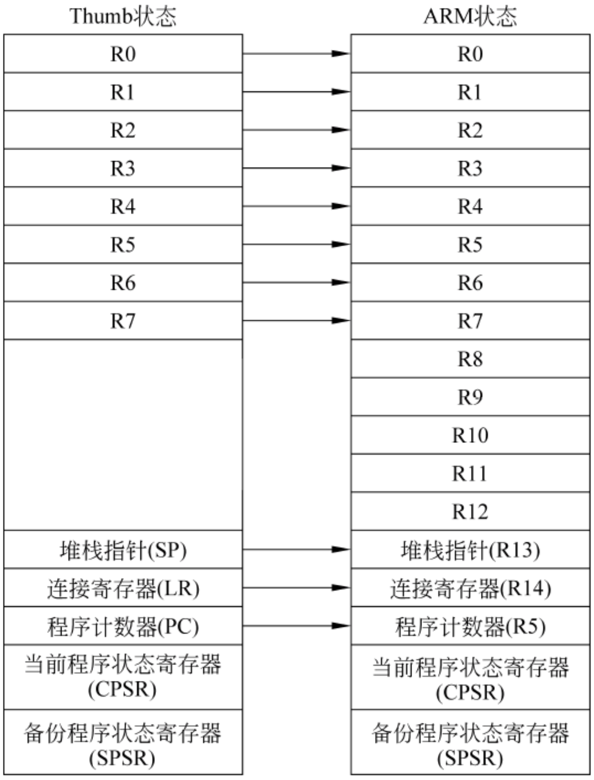


图 3-1　Thumb 寄存器在 ARM 寄存器上的映射

Thumb 状态寄存器与 ARM 状态寄存器有如下关系。

(1) Thumb 状态 R0～R7 与 ARM 状态 R0～R7 相同。

(2) Thumb 状态 CPSR 和 SPSR 与 ARM 状态 CPSR 和 SPSR 相同。

(3) Thumb 状态 SP 映射到 ARM 状态 R13。

(4) Thumb 状态 LR 映射到 ARM 状态 R14。

(5) Thumb 状态 PC 映射到 ARM 状态 PC(R15)。

在程序中可使用 BX 指令实现 ARM 状态和 Thumb 状态的切换。

3.2 ARM汇编语言和汇编语言编程规范

问题：如何编写ARM汇编语言源程序？语句格式如何？用到哪些伪指令或伪操作及运算符？ARM指令分为哪几类？指令条件码是什么，主要有哪些？ARM寻址方式有哪些？Thumb指令有哪些？如何定义符号名称？如何实现ARM汇编与C/C++的混合编程？

重点：ARM指令及其寻址方式。

内容：ARM汇编语句格式，伪操作，伪指令，程序结构，ARM汇编及C/C++的混合编程。

3.2.1 ARM汇编语言语句格式

任务：了解ARM汇编语言源程序的组成，掌握ARM汇编语句格式。

1. ARM汇编语言源程序的组成

汇编语言源程序由若干语句组成，通常，这些语句可以分为3类：指令语句、宏指令语句和伪指令语句。

（1）指令语句

汇编指令是用助记符表示的机器指令，所以又称为机器指令语句，它们由汇编程序汇编成相应的能被CPU直接识别并执行的目标代码，也称为机器代码。例如3.1节中介绍的ARM指令系统中的所有指令均属于机器指令语句。

（2）宏指令语句

在汇编语言中，允许用户为多次重复使用的程序段指定一个名字，然后在程序中用这个名字代替该程序段，定义的过程称为宏定义，该程序段称为宏，只要在相应的位置使用宏名就相当于使用了这段程序。因此，宏指令语句就是宏的引用，宏的引用就是宏指令语句。汇编程序遇到宏指令语句时将其还原成对应的机器指令。

指令语句和宏指令语句都是指令性语句。

（3）伪指令语句

在ARM汇编语言程序中有一些特殊的指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪指令，它们所完成的操作称为伪操作。

伪指令语句是一种指示性语句，用于向汇编程序提供汇编过程中要求的一些辅助信息，如给变量分配内存单元地址、定义各种符号、实现分段等。

伪指令语句与指令性语句最大的区别为：

① 伪指令语句经过汇编后不产生任何机器代码，而指令性语句经汇编后产生相应的机器代码。

② 伪指令语句所指示的操作在汇编程序时就完成了，而指令性语句的操作必须在程序运行时才能完成。

2. ARM汇编语句的格式

ARM汇编语言语句格式如下所示：

```
{symbol}{instruction|directive|pseudo-instruction}{;comment}
```

其中：

(1) symbol 代表符号。在 ARM 汇编语言中，符号必须从一行的行头开始，并且符号中不能包含空格。在指令和伪指令中符号用做地址标号(label)；在有些指示符中，符号用做变量或常量。

(2) instruction 代表指令。在 ARM 汇编语言中，指令不能从一行的行头开始。在一行语句中，指令的前面必须有空格或者符号。

(3) directive 代表指示符。

(4) pseudo-instruction 代表伪指令。

(5) comment 代表语句的注释。在 ARM 汇编语言中注释以分号;开头。注释的结尾即为一行的结尾。注释也可以单独占用一行。

在 ARM 汇编程序中，所有标号在一行中必须顶格书写，其后面不要添加冒号:。而所有指令均不能顶格书写。ARM 汇编器对标识符大小写敏感(即区分大小写字母)，书写标号及指令时字母大小写要一致。在 ARM 汇编程序中，ARM 指令、伪指令、寄存器名可以全部为大写字母，也可以全部为小写字母，但不要大小写混合使用。在源程序中的语句之间可以插入空行，以使得源代码的可读性更好。

如果单行代码太长，可以使用字符\将其分行。\后不能有任何字符，包括空格和制表符等。

对于变量的设置、常量的定义，其标识符在一行中必须顶格书写。

3.2.2 ARM 汇编器的伪操作

任务：了解什么是 ARM 汇编语言伪操作，掌握 ARM 汇编器的各种伪操作的使用方法。

在 ARM 的汇编程序中，伪操作主要有符号定义伪操作、数据定义伪操作、汇编控制伪操作及其他常用的伪操作等。

1. 符号定义伪操作

符号定义伪操作用于定义 ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等。常见的符号定义伪操作有如下几种。

① 用于定义全局变量的 GBLA、GBLL 和 GBLS。

② 用于定义局部变量的 LCLA、LCLL 和 LCLS。

③ 用于对变量赋值的 SETA、SETL、SETS。

④ 为通用寄存器列表定义名称的 RLIST。

(1) GBLA、GBLL 和 GBLS

语法格式：GBLA(GBLL 或 GBLS)全局变量名

功能说明：GBLA、GBLL 和 GBLS 伪指令用于定义一个 ARM 程序中的全局变量，并将其初始化。其中：

① GBLA 伪操作用于定义一个全局的数字变量，并初始化为 0。

② GBLL 伪操作用于定义一个全局的逻辑变量，并初始化为 F(假)。

③ GBLS 伪操作用于定义一个全局的字符串变量，并初始化为空。

由于以上 3 条伪操作指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。

范例：

```
GBLA Test1                    ;定义一个全局的数字变量,变量名为 Test1
Test1 SETA 0xaa               ;将该变量赋值为 0xaa
GBLL Test2                    ;定义一个全局的逻辑变量,变量名为 Test2
Test2 SETL {TRUE}             ;将该变量赋值为真
GBLS Test3                    ;定义一个全局的字符串变量,变量名为 Test3
Test3 SETS"Testing"           ;将该变量赋值为"Testing"
```

(2) LCLA、LCLL 和 LCLS

语法格式：LCLA(LCLL 或 LCLS)局部变量名

功能说明：LCLA、LCLL 和 LCLS 伪操作用于定义一个 ARM 程序中的局部变量，并将其初始化。其中：

① LCLA 伪操作用于定义一个局部的数字变量，并初始化为 0。

② LCLL 伪操作用于定义一个局部的逻辑变量，并初始化为 F(假)。

③ LCLS 伪操作用于定义一个局部的字符串变量，并初始化为空。

以上 3 条伪指令用于声明局部变量，在其作用范围内变量名必须唯一。

范例：

```
LCLA Test4                    ;声明一个局部的数字变量,变量名为 Test4
Test4 SETA 0xaa               ;将该变量赋值为 0xaa
LCLL Test5                    ;声明一个局部的逻辑变量,变量名为 Test5
Test5 SETL {TRUE}             ;将该变量赋值为真
LCLS Test6                    ;定义一个局部的字符串变量,变量名为 Test6
Test6 SETS"Testing"           ;将该变量赋值为"Tesing"
```

(3) SETA、SETL、SETS

语法格式：变量名 SETA(SETL 或 SETS)表达式

功能说明：伪操作 SETA、SETL、SETS 用于给一个已经定义的全局变量或局部变量赋值。

① SETA 伪操作用于给一个数学变量赋值。

② SETL 伪操作用于给一个逻辑变量赋值。

③ SETS 伪操作用于给一个字符串变量赋值。

其中，“变量名”为已经定义过的全局变量或局部变量，表达式为将要赋给变量的值。

(4) RLIST

语法格式：名称 RLIST{寄存器列表}

功能说明：RLIST 伪操作可用于对一个通用寄存器列表定义名称，使用该伪操作定义的名称可在 ARM 指令 LDM/STM 中使用。在 LDM/STM 指令中，列表中的寄存器按照寄存器的编号由低到高的顺序进行访问，而与列表中的寄存器排列次序无关。

范例：

```
RegList RLIST {R0-R5,R8,R10}        ;将寄存器列表名称定义为 RegList
```

2. 数据定义伪操作

数据定义(Data Definition)伪操作一般用于为特定的数据分配存储单元，同时可完成已

分配存储单元的初始化。常见的数据定义伪操作有如下几种。

① DCB 用于分配一块连续的字节存储单元并用指定的数据初始化。

② DCW(DCWU)用于分配一块连续的半字存储单元并用指定的数据初始化。

③ DCD(DCDU)用于分配一块连续的字存储单元并用指定的数据初始化。

④ DCFD(DCFDU)用于为双精度的浮点数分配一块连续的字存储单元并用指定的数据初始化。

⑤ DCFS(DCFSU)用于为单精度的浮点数分配一块连续的字存储单元并用指定的数据初始化。

⑥ DCQ(DCQU)用于分配一块以 8 字节为单位的连续的存储单元并用指定的数据初始化。

⑦ SPACE 用于分配一块连续的存储单元。

⑧ MAP 用于定义一个结构化的内存表首地址。

⑨ FIELD 用于定义一个结构化的内存表的数据域。

(1) DCB

语法格式：标号 DCB 表达式

功能说明：DCB 伪操作用于分配一块连续的字节存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为 0～255 的数字或字符串。DCB 也可用＝代替。

范例：

```
Str DCB"This is a test"          ;分配一块连续的字节存储单元并初始化
```

(2) DCW(或 DCWU)

语法格式：标号 DCW(或 DCWU) 表达式

功能说明：DCW(或 DCWU)伪操作用于分配一块连续的半字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。

用 DCW 分配的字存储单元是半字对齐的，而用 DCWU 分配的字存储单元并不严格半字对齐。

范例：

```
DataTest DCW 1,2,3               ;分配一块连续的半字存储单元并初始化
```

(3) DCD(或 DCDU)

语法格式：标号 DCD(或 DCDU)表达式

功能说明：DCD(或 DCDU)伪操作用于分配一块连续的字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。DCD 也可用 & 代替。

用 DCD 分配的字存储单元是字对齐的，而用 DCDU 分配的字存储单元并不严格字对齐。

范例：

```
DataTest DCD 4,5,6               ;分配一块连续的字存储单元并初始化
```

(4) DCFD(或 DCFDU)

语法格式：标号 DCFD(或 DCFDU)表达式

功能说明：DCFD(或 DCFDU)伪操作用于为双精度的浮点数分配一块连续的字存储

单元并用伪指令中指定的表达式初始化。每个双精度的浮点数占据两个字单元。

用 DCFD 分配的字存储单元是字对齐的，而用 DCFDU 分配的字存储单元并不严格字对齐。

范例：

```
FDataTest DCFD 2E115,-5E7          ;分配一块连续的字存储单元并初始化为指定的双精度数
```

(5) DCFS(或 DCFSU)

语法格式：标号 DCFS(或 DCFSU)表达式

功能说明：DCFS(或 DCFSU)伪操作用于为单精度的浮点数分配一块连续的字存储单元并用伪指令中指定的表达式初始化。每个单精度的浮点数占据一个字单元。

用 DCFS 分配的字存储单元是字对齐的，而用 DCFSU 分配的字存储单元并不严格字对齐。

范例：

```
FDataTest DCFS 2E5,-5E7            ;分配一块连续的字存储单元并初始化为指定的单精度数
```

(6) DCQ(或 DCQU)

语法格式：标号 DCQ(或 DCQU)表达式

功能说明：DCQ (或 DCQU)伪操作用于分配一块以 8 个字节为单位的连续存储区域并用伪操作指定的表达式初始化。

用 DCQ 分配的存储单元是字对齐的，而用 DCQU 分配的存储单元并不严格字对齐。

范例：

```
DataTest DCQ 100                   ;分配一块连续的存储单元并初始化为指定的值
```

(7) SPACE

语法格式：标号 SPACE 表达式

功能说明：SPACE 伪操作用于分配一块连续的存储区域并初始化为 0。其中，表达式为要分配的字节数。SPACE 也可用％代替。

范例：

```
DataSpace SPACE 100                ;分配连续的 100 字节的存储单元并初始化为 0
```

(8) MAP

语法格式：MAP 表达式{,基址寄存器}

功能说明：MAP 伪操作用于定义一个结构化的内存表的首地址。MAP 也可用^代替。

表达式可以为程序中的标号或数学表达式，基址寄存器为可选项，当基址寄存器选项不存在时，表达式的值即为内存表的首地址，当该选项存在时，内存表首地址为表达式值与基址寄存器的和。

MAP 伪操作通常与 FIELD 伪操作配合使用来定义结构化的内存表。

范例：

```
MAP 0x100,R0                       ;定义结构化内存表首地址的值为 0x100+R0
```

(9) FILED

语法格式：标号 FIELD 表达式

功能说明：FIELD 伪操作用于定义一个结构化内存表中的数据域。FILED 也可用

＃代替。

表达式的值为当前数据域在内存表中所占的字节数。

FIELD 伪操作通常与 MAP 伪操作配合使用来定义结构化的内存表。MAP 伪操作用于定义内存表首地址，FIELD 伪操作用于定义内存表中的各个数据域，并可为每个数据域指定一个标号供其他指令引用。

注意 MAP 和 FIELD 伪操作仅用于定义数据结构，并不实际分配存储单元。

范例：

```
MAP 0x100                   ;定义结构化内存表首地址的值为 0x100
A FIELD 16                  ;定义 A 的长度为 16 字节，位置为 0x100
B FIELD 32                  ;定义 B 的长度为 32 字节，位置为 0x110
S FIELD 256                 ;定义 S 的长度为 256 字节，位置为 0x130
```

3. 汇编控制伪操作

汇编控制(Assembly Control)伪操作用于控制汇编程序的执行流程，常用的汇编控制伪操作指令包括以下几条。

(1) IF、ELSE、ENDIF

语法格式：

```
IF 逻辑表达式
    指令序列 1
ELSE
    指令序列 2
ENDIF
```

功能说明：IF、ELSE、ENDIF 伪操作能根据条件是否成立决定是否执行某个指令序列。若 IF 后面的逻辑表达式为真，则执行指令序列 1，否则执行指令序列 2。

其中，ELSE 及指令序列 2 可以没有，此时，若 IF 后面的逻辑表达式为真，则执行指令序列 1，否则继续执行后面的指令。

IF、ELSE、ENDIF 伪操作可以嵌套使用。

范例：

```
GBLL Test                   ;声明一个全局的逻辑变量，变量名为 Test
IF Test=TRUE
     指令序列 1
ELSE
     指令序列 2
ENDIF
```

(2) WHILE、WEND

语法格式：

```
WHILE 逻辑表达式
指令序列
WEND
```

功能说明：WHILE、WEND 伪操作能根据条件是否成立决定是否循环执行某个指令

序列。若 WHILE 后面的逻辑表达式为真,则执行指令序列,该指令序列执行完毕后,再判断逻辑表达式的值,若为真,则继续执行,一直到逻辑表达式的值为假。

WHILE、WEND 伪操作可以嵌套使用。

范例:

```
GBLA Counter                  ;声明一个全局的数学变量,变量名为 Counter
Counter SETA 3                ;由变量 Counter 控制循环次数
…
WHILE Counter &lt;10
指令序列
WEND
```

(3) MACRO、MEND

语法格式:

```
MACRO
{$ 标号}宏名{$ 参数 1,$ 参数 2,…}
指令序列
MEND
```

功能说明:MACRO、MEND 伪操作可以将一段代码定义为一个整体,称为宏指令,然后就可以在程序中通过宏指令多次调用该段代码。

其中,$ 标号在宏指令被展开时会被替换为用户定义的符号,宏指令可以使用一个或多个参数,当宏指令被展开时,这些参数被相应的值替换。

宏指令的使用方式和功能与子程序有些相似,子程序可以提供模块化的程序设计、节省存储空间并提高运行速度,但在使用子程序结构时需要保护现场,从而增加了系统的开销,因此,在代码较短且需要传递的参数较多时,可以使用宏指令代替子程序。

包含在 MACRO 和 MEND 之间的指令序列称为宏定义体,在宏定义体的第一行应声明宏的原型(包含宏名、所需的参数),然后就可以在汇编程序中通过宏名来调用该指令序列。

在源程序被编译时,汇编器将宏调用展开,用宏定义中的指令序列代替程序中的宏调用,并将实际参数的值传递给宏定义中的形式参数。

MACRO、MEND 伪指令可以嵌套使用。

范例:

```
MACRO                         ;宏定义开始
$label xmac $p1,$p2           ;宏的名称为 xmac,有两个参数$p1,$p2
                              ;宏的标号$label 可用于构造宏定义体内的其他标号名称
;code
$label.loop1                  ;$label.loop1 为宏定义体的内部标号
;code
$label.loop2                  ;$label.loop2 为宏定义体的内部标号
BL $sp1                       ;参数$sp1 为一个子程序的名称
BGT $label.loop2
;code
ADR R0,$p2
;code
```

```
MEND                          ;宏定义结束
;在程序中调用该宏
abc xmac subrl,de             ;通过宏的名称 xmac 调用宏,其中宏的标号为 abc
                              ;参数 1 为 subrl,参数 2 为 de
;程序被汇编后,宏展开的结果
;code
abcloop1                      ;用标号$label 的实际值 abc 代替$label 构成标号 abcloop1
;code
BGT abcloop1
abcloop2
;code
BL subrl                      ;参数 1 的实际值为 subrl
BGT abcloop2
;code
ADR R0,de                     ;参数 2 的实际值为 de
```

(4) MEXIT

语法格式：MEXIT

功能说明：用于从宏定义中跳转出去。

范例：

```
MACRO
$abc macroabc $param1,$param2
;code
WHILE condition1
;code
IF condition2
;code
MEXIT                         ;从宏中跳转出去
ELSE
;code
ENDIF
WEND
;code
MEND
```

4. 其他常用的伪操作

还有一些其他的伪操作,在汇编程序中经常使用的包括以下几种。

(1) AREA

语法格式：AREA 段名　属性 1,属性 2,…

功能说明：用于定义一个代码段或数据段。其中,段名若以数字开头,则该段名需用|括起来,如|1_test|。

属性字段表示该代码段(或数据段)的相关属性,多个属性间用逗号分隔。常用的属性如下。

① CODE 属性：用于定义代码段,默认为 READONLY。

② DATA 属性：用于定义数据段,默认为 READWRITE。

③ READONLY 属性：指定本段为只读,代码段的默认属性为 READONLY。

④ READWRITE属性：指定本段为可读可写，数据段的默认属性为READWRITE。

⑤ ALIGN属性：使用方式为ALIGN表达式。在默认情况下，ELF(可执行连接文件)的代码段和数据段是按字对齐的，表达式的取值范围为0～31，相应的对齐方式为2的表达式次幂。

⑥ COMMON属性：定义一个通用的段，不包含任何的用户代码和数据。各源文件中同名的COMMON段共享同一段存储单元。

一个汇编语言程序至少要包含一个段，当程序很长时，也可以将程序分为多个代码段和数据段。

范例：

```
AREA Init,CODE,READONLY
```

AREA定义了一个代码段，段名为Init，属性为只读。

(2) ALIGN

语法格式：ALIGN{表达式{,偏移量}}

功能说明：可通过添加填充字节的方式，使当前位置满足一定的对齐方式。其中，表达式的值用于指定对齐方式，可能的取值为2的幂，如1、2、4、8、16等。

若未指定表达式，则将当前位置对齐到下一个字的位置。偏移量也为一个数字表达式，若使用该字段，则当前位置的对齐方式为：2的表达式次幂＋偏移量。

范例：

```
AREA Init,CODE,READONLY,ALIGN=3              ;指定后面的指令为8字节对齐
指令序列
END
```

(3) CODE16、CODE32

语法格式：CODE16(或CODE32)

功能说明：

CODE16伪操作用于通知编译器，其后的指令序列为16位的Thumb指令。

CODE32伪操作用于通知编译器，其后的指令序列为32位的ARM指令。

若在汇编源程序中同时包含ARM指令和Thumb指令，可用CODE16伪指令通知编译器其后的指令序列为16位的Thumb指令，用CODE32伪操作通知编译器其后的指令序列为32位的ARM指令。因此，在使用ARM指令和Thumb指令混合编写的代码里，可用这两条伪操作进行切换，但注意它们只通知编译器其后指令的类型，并不能对处理器进行状态的切换。

范例：

```
AREA Init,CODE,READONLY
…
CODE32                    ;通知编译器其后的指令为32位的ARM指令
LDR R0,=NEXT+1            ;将跳转地址放入寄存器R0
BX R0                     ;程序跳转到新的位置执行,并将处理器切换到Thumb工作状态
…
CODE16                           ;通知编译器其后的指令为16位的Thumb指令
```

```
NEXT LDR R3,=0x3FF
…
END                                   ;程序结束
```

(4) ENTRY

语法格式：ENTRY

功能说明：ENTRY 伪操作用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个 ENTRY(也可以有多个，当有多个 ENTRY 时，程序的真正入口点由链接器指定)，但在一个源文件中最多只能有一个 ENTRY(可以没有)。

范例：

```
AREA Init,CODE,READONLY
ENTRY                                 ;指定应用程序的入口点
…
```

(5) END

语法格式：END

功能说明：END 伪操作用于通知编译器已经到了源程序的结尾。

范例：

```
AREA Init,CODE,READONLY
…
END                                   ;指定应用程序编译结束
```

(6) EQU

语法格式：名称 EQU 表达式{,类型}

功能说明：EQU 伪操作用于为程序中的常量、标号等定义一个等效的字符名称，类似于 C 语言中的 #define。其中 EQU 可用 * 代替。

“名称”为 EQU 伪操作定义的字符名称，当表达式为 32 位的常量时，可以指定表达式的数据类型，可以有 3 种类型：CODE16、CODE32 和 DATA。

范例：

```
Test EQU 50                           ;定义标号 Test 的值为 50
Addr EQU 0x55,CODE32                  ;定义 Addr 的值为 0x55,且该处为 32 位的 ARM 指令
```

(7) EXPORT(或 GLOBAL)

语法格式：EXPORT 标号{[WEAK]}

功能说明：EXPORT 伪操作用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。

EXPORT 可用 GLOBAL 代替。标号在程序中区分大小写，[WEAK]选项声明其他的同名标号优先于该标号被引用。

范例：

```
AREA Init,CODE,READONLY
EXPORT Stest                          ;声明一个可全局引用的标号 Stest
END
```

(8) IMPORT

语法格式：IMPORT 标号{[WEAK]}

功能说明：IMPORT 伪操作用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，而且无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为 0，若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。

范例：

```
AREA Init,CODE,READONLY
IMPORT Main                    ;通知编译器当前文件要引用标号 Main,但 Main 在其他源文件中定义
END
```

(9) EXTERN

语法格式：EXTERN 标号{[WEAK]}

功能说明：EXTERN 伪操作用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为 0，若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。

范例：

```
AREA Init,CODE,READONLY
EXTERN Main                    ;通知编译器当前文件要引用标号 Main,但 Main 在其他源文件中定义
END
```

(10) GET(或 INCLUDE)

语法格式：GET 文件名

功能说明：GET 伪操作用于将一个源文件包含到当前的源文件中，并在当前位置对被包含的源文件进行汇编处理。可以使用 INCLUDE 代替 GET。

在汇编程序中通常在某源文件中定义一些宏指令，用 EQU 定义常量的符号名称，用 MAP 和 FIELD 定义结构化的数据类型，然后用 GET 伪指令将这个源文件包含到其他的源文件中，与 C 语言中的 include 的作用相似。

GET 伪指令只能用于包含源文件，包含目标文件需要使用 INCBIN 伪指令。

范例：

```
AREA Init,CODE,READONLY
GET a1.s                       ;通知编译器当前源文件包含源文件 a1.s
GET C:\a2.s                    ;通知编译器当前源文件包含源文件 C:\ a2.s
END
```

(11) INCBIN

语法格式：INCBIN 文件名

功能说明：INCBIN 伪操作用于将一个目标文件或数据文件包含到当前的源文件中，

对被包含的文件不做任何变动，直接存放在当前文件中，编译器从其后开始继续处理。

范例：

```
AREA Init,CODE,READONLY
INCBIN a1.dat                    ;通知编译器当前源文件包含文件 a1.dat
INCBIN C:\a2.txt                 ;通知编译器当前源文件包含文件 C:\a2.txt
END
```

(12) RN

语法格式：名称 RN 表达式

功能说明：RN 伪指令用于给一个寄存器定义一个别名。采用这种方式可以方便程序员记忆该寄存器的功能。其中，名称为给寄存器定义的别名，表达式为寄存器的编码。

范例：

```
Temp RN R0                       ;为 R0 定义一个别名 Temp
```

(13) ROUT

语法格式：{名称}ROUT

功能说明：ROUT 伪操作用于给一个局部变量定义作用范围。在程序中未用该伪指令时，局部变量的作用范围为所在 AREA，而用 ROUT 后，局部变量的作用范围为当前 ROUT 和下一 ROUT 之间。

3.2.3　ARM 汇编器支持的伪指令

任务：了解什么是 ARM 汇编语言伪指令，掌握 ARM 汇编器的各种伪指令的使用方法。

ARM 中的伪指令不是真正的 ARM 指令或者 Thumb 指令，这些伪指令在汇编编译时被替换成对应的 ARM 或 Thumb 指令(序列)。ARM 伪指令包括 ADR、ADRL、LDR 和 NOP 等。

1. ADR(小范围的地址读取伪指令)

语法格式：

```
ADR{cond}register,expr
```

其中，cond 为可选的指令执行的条件，register 为目标寄存器，expr 为基于 PC 或者基于寄存器的地址表达式，其取值范围如下。

(1) 当地址值不是字对齐时，其取值范围为－255～255。

(2) 当地址值是字对齐时，其取值范围为－1020～1020。

(3) 当地址值是 16 字节对齐时，其取值范围将更大。

功能说明：将基于 PC 的地址值或基于寄存器的地址值读取到寄存器中。

在汇编编译器处理源程序时，ADR 伪指令被编译器替换成一条合适的指令。通常，编译器用一条 ADD 指令或 SUB 指令来实现该 ADR 伪指令的功能。

因为 ADR 伪指令中的地址是基于 PC 或者基于寄存器的，所以 ADR 读取到的地址为位置无关的地址。若 ADR 伪指令中的地址是基于 PC 的，则该地址与 ADR 伪指令必须在同一个代码段中。

范例：

```
start MOV R0,#10                ;因为 PC 值为当前指令地址值加 8 字节
ADR R4,start                    ;本 ADR 伪指令将被编译器替换成 SUB R4,PC,#0xc
```

2. ADRL(中等范围的地址读取伪指令)

语法格式：ADRL{cond}register,expr

功能说明：基于 PC 或基于寄存器的地址值读取到寄存器中。ADRL 伪指令比 ADR 伪指令可以读取更大范围的地址。ADRL 伪指令在汇编时被编译器替换成两条指令，即使一条指令可以完成该伪指令的功能。

范例：

```
start MOV R0,#10                        ;因为 PC 值为当前指令地址值加 8 字节
ADRL R4,start+60000                     ;本 ADRL 伪指令将被编译器替换成下面两条指令
ADD R4,PC,#0xe800,ADD R4,R4,#0x254
```

3. LDR(大范围的地址读取伪指令)

语法格式：

```
LDR{cond}register,=[expr|label-expr]
```

其中，expr 为 32 位的常量。

编译器将根据 expr 的取值情况按如下方式处理 LDR 伪指令。

(1) 当 expr 表示的地址值没有超过 MOV 或 MVN 指令中地址的取值范围时，编译器用合适的 MOV 或 MVN 指令代替该 LDR 伪指令。

(2) 当 expr 表示的地址值超过 MOV 或者 MVN 指令中地址的取值范围时，编译器将该常数放在数据缓冲区中，同时用一条基于 PC 的 LDR 指令读取该常数。

label-expr 为基于 PC 的地址表达式或者是外部表达式。当 label-expr 为基于 PC 的地址表达式时，编译器将 label-expr 表示的数值放在数据缓冲区(literal pool)中，然后将该 LDR 伪指令处理成一条基于 PC 到该数据缓冲区单元(即计算出数据缓冲区所在的地址相对 PC 的偏移距离，用于最终计算出数据缓冲区所在的地址)的 LDR 指令，从而通过间接寻址的形式将 label-expr 读取到寄存器中。这时，要求该数据缓冲区单元到 PC 的偏移量小于 4KB。当 label-expr 为外部表达式，或者非当前段的表达式时，汇编编译器将在目标文件中插入一个地址重定位伪操作，这样连接器将在连接时生成该地址。

功能说明：将一个 32 位的常数或者一个地址值读取到寄存器中。

LDR 伪指令主要有以下两种用途。

(1) 当需要读取到寄存器中的数据超过了 MOV 及 MVN 指令可以操作的范围时，可以使用 LDR 伪指令将该数据读取到寄存器中。

(2) 将一个基于 PC 的地址值或者外部的地址值读取到寄存器中。

由于这种地址值是在连接时确定的，所以这种代码不是位置无关的。同时 LDR 伪指令的 PC 值到数据缓冲区中的目标数据所在的地址的偏移量要小于 4KB。

范例 1：将 0xFF0 读取到 R1 中

```
LDR R1,=0xFF0
```

汇编后将得到：

```
MOV R1,0xFF0
```

范例 2：将 0xFFF 读取到 R1 中

```
LDR R1,=0xFFF
```

汇编后将得到：

```
LDR R1,[PC,OFFSET_TO_LPOOL]
...
LPOOL DCD 0xFFF
```

范例 3：将外部地址 ADDR1 读取到 R1 中

```
LDR R1,=ADDR1
```

汇编后将得到：

```
LDR R1,[PC,OFFSET_TO_LPOOL]
...
LPOOL DCD ADDR1
```

4. NOP 空操作伪指令

语法格式：NOP

功能说明：在汇编时将被替换成 ARM 中的空操作。

范例：

```
MOV R0,R0
```

NOP 伪指令不影响 CPSR 中的条件标志位。

3.3 ARM汇编语言程序格式

问题： ARM 指令分为哪几类？指令条件码是什么，主要有哪些？ARM 寻址方式有哪些？Thumb 指令有哪些？

重点： ARM 指令及其寻址方式。

内容： ARM 指令系统，ARM 寻址方式，ARM 指令集，Thumb 指令集。

3.3.1 ARM汇编语言程序中常用的符号

任务： 掌握 ARM 汇编语言中常用符号的定义规则，掌握程序中的变量、常量、变量替换方法、标号和局部标号的使用方法。

在 ARM 汇编语言程序设计中，经常使用各种符号代替地址、变量和常量，以增强程序的可读性。尽管符号由编程者命名，但并不是任意的，必须遵循以下约定。

(1) 符号由大小写字母、数字和下划线组成。

(2) 符号区分大小写,同名的大、小写符号会被编译器当成是两个不同的符号。

(3) 符号在其作用范围内必须唯一。

(4) 自定义的符号名不能与系统的保留字相同。

(5) 符号名不应与指令或伪指令同名。

1. 程序中的变量

程序中的变量是指其值在程序的运行过程中可以改变的量。ARM(Thumb)汇编程序所支持的变量有数字变量、逻辑变量和字符串变量。

数字变量用于在程序运行过程中保存数值,但注意数字值的大小不应超出数字变量所能表示的范围。

逻辑变量用于在程序运行过程中保存逻辑值,逻辑值只有两种取值:真和假。

字符串变量用于在程序运行过程中保存一个字符串,但注意字符串的长度不应超出字符串变量所能表示的范围。

在 ARM(Thumb)汇编语言程序设计中,可使用 GBLA、GBLL、GBLS 伪操作声明全局变量,使用 LCLA、LCLL、LCLS 伪操作声明局部变量,并可用 SETA、SETL 和 SETS 对其进行初始化。

2. 程序中的常量

程序中的常量是指其值在程序的运行过程中不能被改变的量。ARM(Thumb)汇编程序所支持的常量有数字常量、逻辑常量和字符串常量。

数字常量一般为 32 位的整数,当作为无符号数时,其取值范围为 $0\sim2^{32}-1$;当作为有符号数时,其取值范围为 $-2^{31}\sim2^{31}-1$。

逻辑常量只有两种取值:真和假。

字符串常量为一个固定的字符串,一般用于程序运行时给出信息提示。

3. 程序中的变量替换

程序中的变量可通过替换操作取得一个常量。替换操作符为 $。

如果在数字变量前面有一个替换操作符 $,编译器会将该数字变量的值转换为十六进制的字符串,并用该十六进制的字符串替换 $ 后的数字变量。

如果在逻辑变量前面有一个替换操作符 $,编译器会将该逻辑变量替换为它的取值(真或假)。

如果在字符串变量前面有一个替换操作符 $,编译器会用该字符串变量的值替换 $ 后的字符串变量。

范例:

```
LCLS S1                          ;定义局部字符串变量 S1 和 S2
LCLS S2
S1 SETS"Test!"
S2 SETS"This is a $S1"           ;字符串变量 S2 的值为"This is a Test!"
```

如果在程序中需要使用字符 $,则用 $$ 来表示,编译器将不进行变量替换,而是将 $$ 当做 $。

4. 标号

标号是表示程序中的指令或者数据地址的符号，根据标号的生成方式可分为以下 3 种。

(1) 基于 PC 的标号

基于 PC 的标号是位于指令前或者程序中数据定义伪操作前的标号。这种标号在汇编时将被处理成 PC 值加上(或减去)一个数字常量，常用于表示跳转指令的目标地址，或者代码段中所嵌入的少量数据。

(2) 基于寄存器的标号

基于寄存器的标号通常用 MAP 和 FILED 伪操作定义，也可以用 EQU 伪操作定义，这种标号在汇编时将被处理成寄存器的值加上(或减去)一个数字常量，常用于访问位于数据段中的数据。

(3) 绝对地址

绝对地址是一个 32 位的数字量，它可以寻址的范围为 $0\sim2^{32}-1$，即可以直接寻址整个内存空间。

5. 局部标号

局部标号主要在局部范围内使用，它由两部分组成：开头是一个 0～99 之间的数字，后面紧接一个通常表示该局部变量作用范围的符号。

局部变量的作用范围通常为当前段，也可用伪操作 ROUT 来定义局部变量的作用范围。

局部变量定义的语法格式为：

```
N(routname)
```

其中：

(1) N 为 0～99 之间的数字。

(2) routname 为符号，通常为该变量作用范围的名称(用 ROUT 伪操作定义的)。

局部变量应用的语法格式为：

```
%{F|B}{A|T} N{routname}
```

其中：

(1) N 为局部变量的数字号。

(2) routname 为当前作用范围的名称(用 ROUT 伪操作定义的)。

(3) %表示引用操作。

(4) F 指示编译器只向前搜索。

(5) B 指示编译器只向后搜索。

(6) A 指示编译器搜索宏的所有嵌套层次。

(7) T 只是编译器搜索宏的当前层次。

如果 F 和 B 都没有指定，编译器先向前搜索，再向后搜索。如果 A 和 T 都没有指定，编译器将从当前层次搜索到宏的最高层次，比当前层次低的层次不再搜索。

如果指定了 routname，编译器将向前搜索最近的 ROUT 伪操作，若 routname 与该 ROUT 伪操作定义的名称不匹配，编译器报告错误，编译失败。

3.3.2 汇编语言程序中的表达式和运算符

任务：掌握ARM汇编语言中各种表达式和运算符的使用方法。

在汇编语言程序设计中，也经常使用各种表达式，表达式一般由变量、常量、运算符和括号构成。常用的表达式有数字表达式、逻辑表达式和字符串表达式，其运算次序和优先级遵循以下规则。

（1）优先级相同的双目运算符的运算顺序为从左到右。

（2）相邻的单目运算符的运算顺序为从右到左，且单目运算符的优先级高于其他运算符。

（3）括号运算符的优先级最高。

1. 数字表达式及运算符

数字表达式一般由数字常量、数字变量、数字运算符和括号构成。相关的运算符如下。

（1）+、−、×、/及MOD算术运算符

以X和Y表示两个数字表达式，以上的算术运算符代表的运算如下。

① X+Y表示X与Y相加。

② X−Y表示X与Y相减。

③ X×Y表示X与Y相乘。

④ X/Y表示X除以Y。

⑤ X:MOD:Y表示求X除以Y的余数。

（2）ROL、ROR、SHL及SHR移位运算符

以X和Y表示两个数字表达式，以上的移位运算符代表的运算如下。

① X:ROL:Y表示将X循环左移Y位。

② X:ROR:Y表示将X循环右移Y位。

③ X:SHL:Y表示将X左移Y位。

④ X:SHR:Y表示将X右移Y位。

（3）AND、OR、NOT及EOR按位逻辑运算符

以X和Y表示两个数字表达式，以上的按位逻辑运算符代表的运算如下。

① X:AND:Y表示将X和Y按位进行逻辑与的操作。

② X:OR:Y表示将X和Y按位进行逻辑或的操作。

③ :NOT:Y表示将Y按位进行逻辑非的操作。

④ X:EOR:Y表示将X和Y按位进行逻辑异或的操作。

2. 逻辑表达式及运算符

逻辑表达式一般由逻辑量、逻辑运算符和括号构成，逻辑表达式的运算结果为真或假。与逻辑表达式相关的运算符如下。

（1）=、>、<、>=、<=、/=、<>运算符

以X和Y表示两个逻辑表达式，以上的逻辑运算符代表的运算如下。

① X=Y表示X等于Y。

② X＞Y 表示 X 大于 Y。

③ X＜Y 表示 X 小于 Y。

④ X＞＝Y 表示 X 大于等于 Y。

⑤ X＜＝Y 表示 X 小于等于 Y。

⑥ X/＝Y 表示 X 不等于 Y。

⑦ X＜＞Y 表示 X 不等于 Y。

(2) LAND、LOR、LNOT 及 LEOR 运算符

以 X 和 Y 表示两个逻辑表达式，以上的逻辑运算符代表的运算如下。

① X:LAND:Y 表示将 X 和 Y 进行逻辑与的操作。

② X:LOR:Y 表示将 X 和 Y 进行逻辑或的操作。

③ :LNOT:Y 表示将 Y 进行逻辑非的操作。

④ X:LEOR:Y 表示将 X 和 Y 进行逻辑异或的操作。

3. 字符串表达式及运算符

字符串表达式一般由字符串常量、字符串变量、运算符和括号构成。编译器所支持的字符串最大长度为 512 字节。常用的与字符串表达式相关的运算符如下。

(1) LEN 运算符

LEN 运算符用于返回字符串的长度(字符数)，以 X 表示字符串表达式，其语法格式如下：

```
:LEN:X
```

(2) CHR 运算符

CHR 运算符用于将 0～255 之间的整数转换为一个字符，以 M 表示某一个整数，其语法格式如下：

```
:CHR:M
```

(3) STR 运算符

STR 运算符用于将一个数字表达式或逻辑表达式转换为一个字符串。对于数字表达式，STR 运算符将其转换为一个以十六进制数组成的字符串；对于逻辑表达式，STR 运算符将其转换为字符串 T 或 F，其语法格式如下：

```
:STR:X
```

其中，X 为一个数字表达式或逻辑表达式。

(4) LEFT 运算符

LEFT 运算符用于返回某个字符串左端的一个子串，其语法格式如下：

```
X:LEFT:Y
```

其中，X 为源字符串，Y 为一个整数，表示要返回的字符个数。

(5) RIGHT 运算符

与 LEFT 运算符相对应，RIGHT 运算符用于返回某个字符串右端的一个子串，其语法

格式如下：

```
X:RIGHT:Y
```

其中，X 为源字符串，Y 为一个整数，表示要返回的字符个数。

（6）CC 运算符

CC 运算符用于将两个字符串连接成一个字符串，其语法格式如下：

```
X:CC:Y
```

其中，X 为源字符串 1，Y 为源字符串 2，CC 运算符将 Y 连接到 X 的后面。

4. 与寄存器和程序计数器（PC）相关的表达式及运算符

常用的与寄存器和程序计数器（PC）相关的表达式及运算符如下。

（1）BASE 运算符

BASE 运算符用于返回基于寄存器的表达式中寄存器的编号，其语法格式如下：

```
:BASE:X
```

其中，X 为与寄存器相关的表达式。

（2）INDEX 运算符

INDEX 运算符用于返回基于寄存器的表达式中相对于其基址寄存器的偏移量，其语法格式如下：

```
:INDEX:X
```

其中，X 为与寄存器相关的表达式。

5. 其他常用运算符

（1）? 运算符

? 运算符用于返回某代码行所生成的可执行代码的长度，例如：

```
? X
```

返回定义符号 X 的代码行所生成的可执行代码的字节数。

（2）DEF 运算符

DEF 运算符用于判断是否定义某个符号，例如：

```
:DEF:X
```

如果符号 X 已经定义，则结果为真，否则为假。

3.3.3 ARM 汇编语言程序的基本结构

任务：理解什么是段，段的分类，可执行映像文件的构成；了解 ARM 汇编语言程序的基本结构。

在 ARM（Thumb）汇编语言程序中，以程序段为单位组织代码。段是相对独立的指令或数据序列，具有特定的名称。段可以分为代码段和数据段，代码段的内容为可执行代码，数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段，当程序较

长时，可以分割为多个代码段和数据段，多个段在程序编译链接时最终形成一个可执行的映像文件。

可执行映像文件通常由以下几部分构成。

(1) 一个或多个代码段，代码段的属性为只读。

(2) 零个或多个包含初始化数据的数据段，数据段的属性为可读写。

(3) 零个或多个不包含初始化数据的数据段，数据段的属性为可读写。

连接器根据系统默认或用户设定的规则，将各个段安排在存储器中的相应位置，因此源程序中段之间的相对位置与可执行的映像文件中段的相对位置一般不会相同。

以下是一个汇编语言源程序的基本结构：

```
AREA Init,CODE,READONLY
ENTRY
Start
LDR R0,=0x3FF5000
LDR R1,0xFF
STR R1,[R0]
LDR R0,=0x3FF5008
LDR R1,0x01
STR R1,[R0]
END
```

在汇编语言程序中，用 AREA 伪操作定义一个段，并说明所定义段的相关属性，本例定义了一个 名为 Init 的代码段，属性为只读。ENTRY 伪操作用来标识程序的入口点，接下来为指令序列，程序的末尾为 END 伪操作，指示编译器源文件结束，每一个汇编程序段都必须有一条 END 伪操作指令，指示代码段结束。

3.3.4 ARM 汇编程序设计举例

任务： 理解本节中的每个程序，并学会使用汇编程序程序设计的 3 种基本结构—顺序结构、分支结构和循环结构进行编程，掌握 ARM 汇编语言子程序调用方法。

同高级语言一样，汇编程序程序设计也有 3 种基本结构：顺序结构、分支结构和循环结构。本节将通过例子来说明各种结构的程序设计方法，并在最后介绍在 ARM 汇编程序中调用子程序的方法。

1. 顺序程序设计举例

通过查表操作将数组中的第 1 项数据和第 5 项数据相加，结果保存到数组中。

分析：程序流程图如图 3-2 所示。

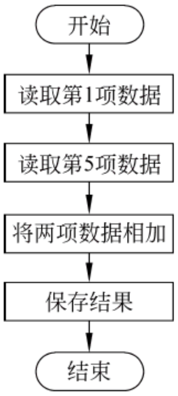


图 3-2 顺序程序设计举例—程序流程图

程序清单：

```
AREA Buf,DATA,READWRITE          ;定义数据段 Buf
Array   DCD 0x11,0x22,0x33,0x44
                                 ;定义 12 个字的数组 Array
        DCD 0x55,0x66,0x77,0x88
        DCD 0x00,0x00,0x00,0x00
```

```
        AREA Example,CODE,READONLY
ENTRY
CODE32
LDR   R0,=Array               ;取得数组 Array 的首地址
LDR   R2,[R0]                 ;装载数组第 1 项字数据给 R2
MOV   R1,#4
LDR   R3,[R0,R1,LSL #2]       ;装载数组第 5 项字数据给 R3
ADD   R2,R2,R3                ;R2 +R3→R2
MOV   R1,#8                   ;R1=8
STR   R2,[R0,R1,LSL #2]       ;保存结果到数组第 9 项
END
```

2. 分支程序设计举例

判读 X、Y 两个变量的大小,从而给变量 Z 赋予不同的值。

分析:程序流程图如图 3-3 所示。

程序清单:

```
…
MOV      R0,#76
MOV      R1,#88
CMP      R0,R1
MOVHI    R2,#100
MOVLS    R2,#50
…
```

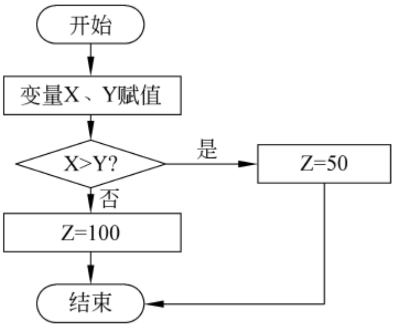


图 3-3 分支程序设计举例—程序流程图

或者:

```
…
    MOV     R0,#76
    MOV     R1,#88
    CMP     R0,R1
    BHI     Next
    MOV     R2,#50
    B       Next2
Next1
    MOV     R2,#100
Next2
…
```

3. 循环程序设计

循环结构由以下两部分组成。

(1) 循环体:要求重复执行的程序段部分。

(2) 循环结束条件:在循环程序中必须给出循环结束条件,否则程序会进入死循环。

在 C 语言中,用 for 和 while 语句实现这两种循环。在汇编语言中,用跳转指令实现这两种循环。

(1) 计算 1+2+3+…+100 的结果。

程序清单:

```
        …
        MOV     R0,#0                       ;初始化 R0=0
        MOV     R2,#1                       ;设置 R2=0,R2 控制循环次数
FOR     CMP     R2,#100                     ;判断 R2 是否小于 100
        BHS     FOR_E                       ;若条件不成立,则退出循环
        ADD     R0,R0,R2                    ;循环体,R0=R0+R2
        ADD     R2,R2,#1                    ;R2=R2+1
        B       FOR
FOR_E   …
```

(2) 编写循环语句实现数据块复制。

程序清单:

```
      LDR     R0,=DATA_DST              ;指向数据目标地址
      LDR     R1,=DATA_SRC              ;指向数据源地址
      MOV     R10,#20                   ;复制数据个数 20×N 个字
                                        ;N 为 LDM 指令操作的数据个数
LOOP  LDMIA   R1!,{R2-R9}               ;从数据源读取 8 个字到 R2～R9
      STMIA   R0!,{R2-R9}               ;将 R2～R9 的数据保存到目标地址
      SUBS    R10,R10,#1                ;R10-1
BNELOOP
```

4. ARM 汇编语言子程序调用

在 ARM 汇编语言程序中,子程序的调用一般是通过 BL 指令来实现的。在程序中使用指令:

```
BL  子程序名
```

即可完成子程序的调用。

通过执行该指令可以完成如下操作:将子程序的返回地址存放在连接寄存器 LR 中,同时将程序计数器 PC 指向子程序的入口点,当子程序执行完毕需要返回调用位置时,只需要将存放在 LR 中的返回地址重新复制给程序计数器 PC。在调用子程序的同时,也可以完成参数的传递和从子程序返回运算的结果,通常可以使用寄存器 R0～R3 完成。

以下是使用 BL 指令调用子程序的汇编语言源程序的基本结构:

```
AREA Init,CODE,READONLY
ENTRY
Start
LDR R0,=0x3FF5000
LDR R1,0xFF
STR R1,[R0]
LDR R0,=0x3FF5008
LDR R1,0x01
STR R1,[R0]
BL PRINT_TEXT
PRINT_TEXT
MOV PC,BL
END
```

3.4 汇编语言与C/C++ 语言的混合编程

问题： ARM 指令分为哪几类？指令条件码是什么，主要有哪些？ARM 寻址方式有哪些？Thumb 指令有哪些？

重点： ARM 指令及其寻址方式。

内容： ARM 指令系统，ARM 寻址方式，ARM 指令集，Thumb 指令集。

ARM 编程可以使用汇编语言和 C/C++ 语言，使用汇编语言编程目标代码执行效率较高，但较为烦琐，设计大型系统时不易维护；而 C/C++ 语言比较简洁明了，但代码即使经过优化，执行效率也比汇编语言的低，特别是在一些实时性强和需要精细处理的场合，C/C++ 语言难以胜任。在一个完整的程序设计中，除了初始化部分用汇编语言完成以外，其主要编程任务一般都用 C/C++ 语言完成。

汇编语言与 C/C++ 的混合编程通常采用以下几种方式。

（1）在 C/C++ 程序中嵌入汇编指令。

（2）在汇编程序和 C/C++ 程序之间进行变量的互访。

（3）汇编程序、C/C++ 程序间的相互调用。

在采用以上几种混合编程技术时，必须遵守一定的调用规则，如物理寄存器的使用、参数的传递等。在实际的编程应用中使用较多的方式是：程序的初始化部分用汇编语言完成，然后用 C/C++ 语言完成主要的编程任务，程序在执行时首先完成初始化过程，然后跳转到 C/C++ 程序中，在汇编程序和 C/C++ 程序之间一般不传递参数，也没有频繁的相互调用，因此，整个程序的结构显得相对简单，容易理解。

3.4.1 在 C/C++ 程序中嵌入汇编指令

任务： 掌握在 C/C++ 程序中嵌入汇编指令的方法。

在 ARM 的 C/C++ 程序中可以使用关键字__asm 来加入一段汇编语言的程序。

语法格式：

```
__asm
    {
        指令[;指令]                /* 注释* /
        …
        [指令]
    }
```

范例：在 ARM 处理器程序中有两个十分常见的函数 enable_IRQ 和 disable_IRQ 用来启用/禁用 IRQ 中断程序。具体的函数源码如下所示。

```
void enable_IRQ(void)              //启用中断程序
  {
     int tmp;                      //定义临时变量,后面使用
      __asm                        //内嵌汇编程序的关键词
```

```
            {
              MRS tmp,CPSR             //将状态寄存器加载给 tmp
              BIC tmp,tmp,#80          //将 IRQ 控制位清 0
              MSR CPSR_c,tmp           //加载程序状态寄存器
            }
        }
        void disable_IRQ(void)         //禁用中断程序
      {
         int tmp;                      //定义临时变量,后面使用
         __asm                         //内嵌汇编程序的关键词
           {
               MRS tmp,CPSR            //将状态寄存器加载给 tmp
               ORR tmp,tmp,#80         //将 IRQ 控制位置 1
               MSR CPSR_c,tmp          //加载程序状态寄存器
           }
    }
```

3.4.2 在 ARM 汇编程序和 C/C++ 程序之间进行变量的互访

任务: 掌握在 ARM 汇编程序和 C/C++ 程序之间进行变量互访的方法,包括 ARM 汇编程序访问 C/C++ 全局变量和 C/C++ 程序访问汇编程序数据。

1. ARM 汇编程序访问 C/C++ 全局变量

汇编程序只能通过地址间接访问 C/C++ 全局变量。要访问全局变量,必须在汇编程序中使用 IMPORT/EXTERN 伪操作引用该全局变量,使用 LDR 伪指令读取该全局变量的内存地址;根据该数据的类型,使用相应的 LDR 指令读取该全局变量;使用相应的 STR 指令存储该全局变量的值。数据类型和 ARM 指令的对应关系如表 3-4 所示。

表 3-4 C/C++ 全局变量的数据类型与 ARM 指令的对应关系

C/C++ 语言中的变量类型	带后缀的 LDR 和 STR 指令	描 述
unsigned char	LDRB/STRB	无符号字符型
unsigned short	LDRH/STRH	无符号短整型
unsigned int	LDR/STR	无符号整型
char	LDRSB/STRSB	字符型(8 位)
short	LDRSH/STRSH	短整型(16 位)

范例:将整型全局变量 globvar 的地址载入 R1,将该地址中包含的值载入 R0,并将其与 2 相加,然后将新值存回 globvar 中。

```
    PRESERVE8
    AREA globals,CODE,READONLY
    EXPORT asmsubroutine
    IMPORT globvar
asmsubroutine
    LDR R1,=globvar;从内存池中读取 globvar,加载到 R1 中
    LDR R0,[R1]
    ADD R0,R0,#2
```

```
STR R0,[R1]
MOV pc,lr
END
```

2. C/C++ 程序访问汇编程序数据

在汇编程序中用 EXPORT/GLOBAL 伪操作声明该符号为全局标号,可以被其他文件应用;在 C/C++ 程序中定义相应数据类型的指针变量;将该指针变量赋值为汇编程序中的全局标号,利用该指针访问汇编程序中的数据。

3.4.3 汇编程序、C/C++ 程序间的相互调用

> **任务:** 掌握 ARM 汇编程序和 C/C++ 程序之间相互调用的方法,包括 C/C++ 程序调用汇编程序和汇编程序调用 C/C++ 程序。

1. C/C++ 程序调用汇编程序

在汇编程序中使用 EXPORT 伪指令声明本子程序可在外部使用,使其他程序可调用该子程序;在 C/C++ 语言程序中使用 extern 关键字声明外部函数,才可以调用此汇编程序的子程序。

范例 1:从 C 程序调用汇编程序,将一个字符串赋值给另一个字符串。

```
#include<stdio.h>
extern void strcopy(char * d,const char * s);
int main()
{
   const char * srcstr="Source string";
   char dststr[]="Destination string";
   /* 下面将 dststr 作为数组进行操作 * /
   printf("Before copying:\n");
   printf("%s\n %s\n",srcstr,dststr);
   strcopy(dststr,srcstr);
   printf("After copying:\n);
   printf("%s\n %s\n",srcstr,dststr);
   return(0);
}
```

下面为调用的汇编程序:

```
     PRESERVE8
     AREA SCopy,CODE,READONLY
     EXPORT strcopy
strcopy                                       ;R0 指向目的字符串,R1 指向源字符串
     LDRB R2,[R1],#1
     STRB R2,[R0],#1
     CMP R2,0
     BNE strcopy
     MOV pc,lr
     END
```

范例 2：从 C++ 中调用汇编程序：

```
struct S
{
    S(int s):i(s){}
    int i;
};

extern"C"void asmfunc(S * );           //声明被调用的汇编函数
int f()
{
    S s(2);                             //初始化结构体 s
    asmfunc(&s);                        //调用汇编子程序 asmfunc
    return s.i * 3;
}
```

下面为调用的汇编程序：

```
    PRESERVE8
    AREA Asm,CODE,READONLY
    EXPORT asmfunc
asmfunc                                ;被调用的汇编程序定义
    LDR R1,[R0]
    ADD R1,R1,#5
    STR R1,[R0]
    MOV pc,lr
    END
```

2. 汇编程序调用 C/C++ 程序

在汇编程序中使用 IMPORT 伪操作声明将要调用的 C/C++ 程序函数。在调用 C/C++ 程序时，要正确设置入口参数，然后使用 BL 指令调用。

范例 1：从汇编程序调用 C 程序

```
//C 函数定义
int g(int a,int b,int c,int d,int e)
{
    return a+b+c+d+e;
}
;汇编程序段的定义。假设程序进入 f 时,R0 中的值为 i
;int f(int i){return g(i,2*i,3*i,4*i,5*i);}
EXPORT f
AREA f,CODE,READONLY
EXPORT g                               ;声明 C 程序 g()
STR lr,[sp,#-4]!                       ;保存返回地址 lr
ADD R1,R0,R0                           ;计算 2*i
ADD R2,R1,R0                           ;计算 3*i
ADD R3,R1,R2                           ;计算 5*i
STR R3,[sp,#-4]!                       ;第 5 个参数通过堆栈传递
ADD R3,R1,R1                           ;计算 4*i
BL g                                   ;调用 C 程序
ADD sp,sp,#4                           ;从堆栈中删除第 5 个参数
```

```
LDR pc,[sp],#4                                ;返回
END
```

范例 2：从汇编程序调用 C++ 程序

```
struct S
{
    S(int s):i(s){}
    int i;
};
extern"C"void cppfunc(S * p)
{
p->i+=5;
}
     AREA Asm,CODE,READONLY
     IMPORT cppfunc                           ;声明被调用的 C++函数名
     EXPORT f
f
     STMFD sp!,{lr}
     MOV R0,#2
     STR R0,[sp,#-4]!                         ;初始化结构体
     MOV R0,sp                                ;调用参数为指向结构体的指针
     BL cppfunc                               ;调用 C++函数 cppfunc
     LDR R0,[sp],#4
     ADD R0,R0,R0,LSL #1
     LDMFD sp!,{pc}
     END
```

小结

本章重点讲述了 ARM 汇编语言程序设计的基本方法，首先介绍了 ARM 及 Thumb 指令集以及 ARM 寻址方式，并通过范例进一步讲解 ARM 指令集的使用方法；然后介绍了 ARM 汇编语言编程规范、程序格式，在编程规范中详细介绍了 ARM 汇编器的伪操作和伪指令，在程序格式中，介绍了 ARM 汇编程序中的常用符号、表达式、运算符和程序的基本结构；最后结合实例介绍了汇编语言与 C/C++ 语言的混合编程方式。

第4章

嵌入式系统硬件设计

学习目标

通过本章的学习，应该掌握：

✍ 握嵌入式最小系统的组成

✍ 握嵌入式系统主要外围部件的工作原理

✍ S3C2440A 启动过程

4.1 嵌入式最小系统

问题：嵌入式系统设计需要考虑哪些部分？这些部分各有什么作用？什么是最小系统法？

重点：实际的嵌入式系统硬件设计过程；嵌入式系统最小系统的组成。

内容：简单介绍嵌入式系统的软件设计和硬件设计的关系、最小系统法以及嵌入式系统最小系统的组成。

在设计一个嵌入式系统产品时，需要重点考虑的因素有两个，一个是嵌入式系统的软件部分，包括了所采用的嵌入式操作系统以及应用程序，用以控制整个嵌入式系统的动作流程；另一个就是嵌入式系统硬件。软硬件系统的设计是互相关联、密不可分的，进行嵌入式系统设计时经常需要在硬件设计和软件设计之间进行权衡与折中。

其中，嵌入式系统硬件部分决定了嵌入式系统本来具有的功能，如运算能力以及扩充功能等，一个好的嵌入式系统需要实现完整的硬件规划才能具备条件达到所需要的功能。在设计好嵌入式系统的硬件架构之后，就要考虑嵌入式系统的软件部分，嵌入式系统软件就像是嵌入式系统的灵魂，决定着所有硬件的操作模式，通过优异的操作系统以及应用程序，可以将有限的硬件装置功能发挥到极限。

一个嵌入式系统开发人员首先必须要了解嵌入式系统的硬件架构，才能够在硬件所提供的有限效率下进行相关的应用程序开发。因此，本章将以 Samsung 公司的 S3C2440A 为例，介绍嵌入式应用系统的硬件设计基本方法和相关部件的原理，包括存储系统以及输入输出装置等。

在实际的嵌入式系统硬件设计过程中，做好需求分析后，要进行体系结构设计，包括软件结构和硬件组成。确定硬件组成时要对用到的微处理器、各种外围设备及接口进行选型，然后再对各个部分进行连接。为了便于进行连接与故障检测，往往采用最小系统法。所谓最小系统法就是通过满足最基本的硬件或者软件环境来开机和运行，从而判断出系统的问题所在。对于硬件，组成最小系统后，如果不能正常启动，可以用好的设备来代替系统中可能有故障的设备，观察故障是否消失。如果能够正常启动，则每次向该系统添加一个设备，以此来判断出现故障的部位。通过不断地添加设备和排除故障，最终实现整个系统的硬件设计。

对于一个典型的嵌入式最小系统，其构成模块及其各部分功能如图 4-1 所示。微处理器本身是不能工作的，必须给它提供电源，加上时钟信号、复位信号，再加上存储器系统，然后微处理器才可能工作。构成嵌入式最小系统时，如果微处理器芯片内没有足够的片内程序存储器，则要外接 SDRAM 进行扩展。在一般情况下，最终的程序都需要固化，因此要用到 Flash 或其他永久性存储介质。

嵌入式最小系统各部分硬件的功能描述如下。

（1）嵌入式微处理器：是系统的工作和控制中心。

（2）电源电路：为微处理器及其他需要电源的外围电路供电。

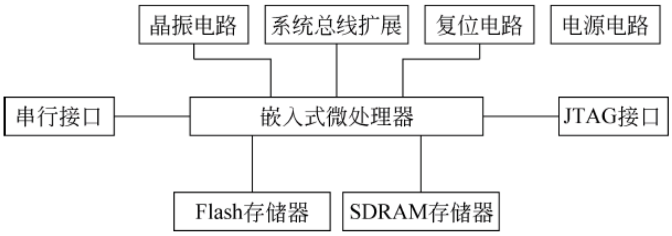


图 4-1　嵌入式最小系统原理图

(3) 晶振电路：为系统提供工作时钟，经由片内 PLL(时钟发生器)电路倍频产生微处理器的工作时钟。

(4) Flash 存储器：存放嵌入式操作系统、用户应用程序或者其他在系统掉电后需要保存的用户数据等。

(5) SDRAM 存储器：作为系统运行时的主要区域，系统及用户数据、堆栈均位于该存储器中。

(6) 串行接口：用于嵌入式系统与其他应用系统间的短距离双向串行通信。

(7) JTAG 接口：可对芯片内部的所有部件进行访问，通过该接口可以对系统进行调试、编程等。

(8) 系统总线扩展：引出地址总线、数据总线和必需的控制总线，便于用户根据自身的特定需求扩展外围电路。

(9) 复位电路：完成系统上电复位和在系统工作时用户按键复位。

在图 4-1 中，用于调试和测试的 JTAG 接口部分在芯片实际工作时不是必需的，但在开发时很重要，所以把这部分也归入最小系统中。

构成最小系统后，若经测试无故障，则逐步添加其他硬件部分，最终形成所需的硬件系统。

因为嵌入式系统追求小体积、低功耗、低成本，所以和设计其他硬件系统一样，一般只考虑添加实际需要的部件。

4.2　S3C2440A 概述

问题：S3C2440A 具有哪些特性？S3C2440A 内包含了哪些片上资源？
重点：S3C2440A 的片上资源。
内容：S3C2440A 特性、片上资源及引脚。

在进行特定应用系统设计之前，有必要了解 ARM 芯片 S3C2440A 及其工作原理。

Samsung 公司的 S3C2440A 的核心是 16/32 位 RISC 微处理器 ARM920T，内含一个由 ARM 公司设计的 16/32 位 ARM9TDMI RISC 处理器核，主频高达 300MHz 或 400MHz。ARM920T 实现了 MMU、AMBA 总线和 Harvard 高速缓冲体系结构，这一结构具有独立的 16KB 的指令 cache 和 16KB 的数据 cache。

S3C2440A 提供了完整的通用系统外围设备,片上资源丰富,性价比高。

S3C2440A 包括如下片上资源。

(1) 16KB 指令 cache 和 16KB 数据 cache。

(2) 1 个 LCD 控制器。

(3) 3 个通道的 UART。

(4) 4 个通道的 DMA 控制器。

(5) 2 通道 SPI。

(6) 8 通道的 10 位 ADC。

(7) 60 个中断源的中断控制器。

(8) 4 通道 PWM 定时器和 1 通道内部定时器。

(9) 16 位的看门狗定时器。

(10) SDRAM 控制器。

(11) SD 接口和 MMC 卡接口。

(12) RTC 模块。

(13) AC'97 解码器接口。

(14) 触摸屏接口。

(15) I2C 总线接口。

(16) I2S 总线接口。

(17) 2 个 USB 主机接口,1 个 USB 设备接口。

(18) 130 个通用 I/O 口和 24 个外部中断端口。

(19) 相机接口。

(20) PLL 时钟发生器。

S3C2440A 共有 289 个引脚,芯片采用 BGA 封装形式,如图 4-2 所示。

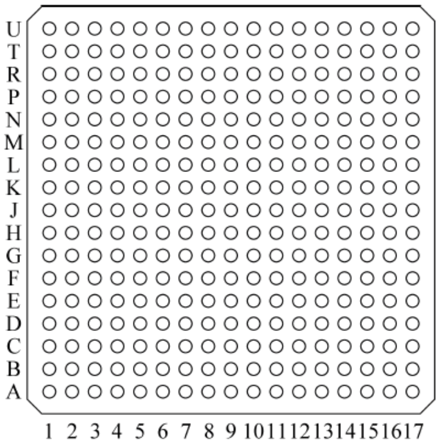


图 4-2 S3C2440A 芯片引脚图

S3C2440A 的引脚以 17×17 的矩阵形式进行排列,每个引脚的标号以字母序号和数字序号的组合形式进行标注。限于篇幅,这里不再对各个引脚进行描述,读者可查阅数据手册。

4.3 S3C2440A 外围部件工作原理

问题: S3C2440A 内包含了哪些外围部件?如何使用这些部件接口?

重点: S3C2440A 芯片内部包含的各个部件接口。

内容: S3C2440A 内的各个部件接口,包括存储器控制器、中断控制器、定时器、LCD 控制器、UART 等。

在 S3C2440A 芯片内部包含了各种外围部件,如存储器控制器、中断控制器、定时器、LCD 控制器、UART 等,片上资源丰富。本节将介绍 S3C2440A 芯片内部包含的各个部件的工作原理。

图 4-3 为 S3C2440A 芯片内部结构图。

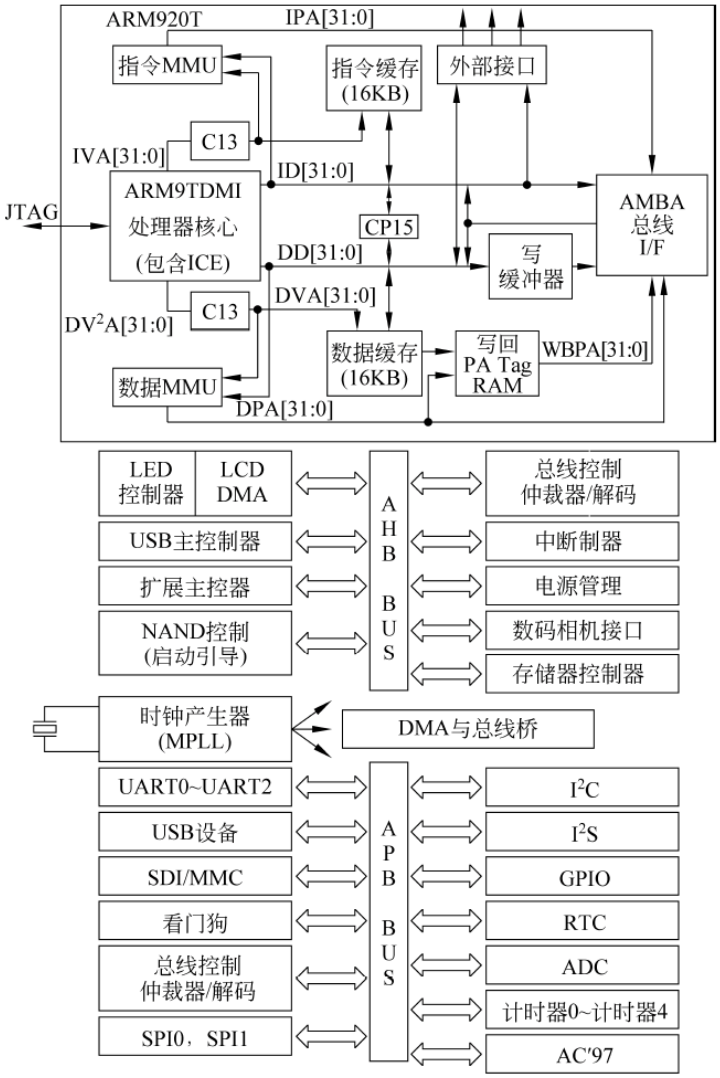


图 4-3 S3C2440A 芯片内部结构

4.3.1 存储器控制器

当前在各类嵌入式系统开发设计中存储模块都是不可或缺的。

任务：掌握 S3C2440A 的存储器控制器的功能、结构及初始化。

1. 存储系统层次结构

简单的嵌入式系统存储结构按作用分两级：寄存器和主存；复杂的嵌入式系统存储结构按作用需要分 4 级：寄存器、cache、主存储器和辅助存储器。

寄存器包含在微处理器内部，用于在指令执行时存放数据。cache 是高速缓存，处理速度介于主存储器和微处理器之间，用于存放主存储器部分内容的副本，以匹配二者的速度，提高系统的整体性能。主存储器也称为内存，是程序和数据的存放区。辅助存储器用于大量数据或程序代码等的备份，其作用与通用计算机的辅助存储器类似。

一般将主存分为 RAM(Random Access Memory)和 ROM(Read Only Memory)。RAM 的特性就是一旦断电,所存储的数据就会全部消失。RAM 又可分成 DRAM(Dynamic RAM)及 SRAM(Static RAM),二者的差异在于数据保存时间。DRAM 需要不断刷新,也就是读出来再写回去,而且还要一个 DRAM 控制器来完成这个动作;而 SRAM 则不需要刷新,而且 SRAM 的存取速度要比 DRAM 快很多。不过 SRAM 比 DRAM 价格昂贵,所以设计的时候会采用折中方案,同时使用两种存储器,将大量的运算处理所需的数据放在 DRAM 里,某些重要的数据放在 SRAM 里。例如,SRAM 常常用于高速缓冲存储器,有更高的速率;DRAM 常常用于嵌入式系统中或者 PC 的主存储器中,有更高的容量和更低的价格。

DRAM 又有普通 DRAM、扩展数据输出 DRAM(Extend Data Output DRAM,EDO DRAM)、同步 DRAM(Synchronous DRAM,SDRAM)、双数据速率 DRAM(Dual Data Rate SDRAM,DDR SDRAM)、Rambus DRAM(简称 RDRAM)等不同的种类。其中,普通 DRAM 的存取速度大致为 60ns,无法与 100MHz 总线匹配;SDRAM 工作在 100MHz 和 133MHz 下,其按额定的频率存取数据,与总线时钟同步,可实现与 100MHz 总线的连接;DDR SDRAM 可以工作在 200MHz 和 266MHz 下,存取容量是 SDRAM 的两倍,对 SDRAM 进行了改进,在相同时钟下,其吞吐量是 SDRAM 的两倍;RAMBUS DRAM 是一种系统级接口,有自己的协议。其速度特别高,最高可达 833MHz,很难大批量生产,因而价格一直居高不下,并且使用 Rambus 时还需要额外支付许可费用。

在嵌入式系统中,通常不提倡采用超前的方案进行设计,SDRAM 的速度和价格都比较适中,所以是嵌入式系统主存储器的最佳选择。

ROM 是只读存储器,其中存储的信息掉电也不会丢失,因而可以利用 ROM 的这种非易失性存储代码和数据。ROM 一般可以分为 EPROM(Erasable-and-Programmable ROM)、PROM(Programmable ROM)、Masked ROM 及 EEPROM(Electrically Erasable-and-Programmable ROM)。这几类存储器都是非易失性的,掉电后也可以永久保存数据。其中,EPROM 就是可以重复擦写的 ROM,不过要用紫外线先清除已经有的数据再写新的数据进去;PROM 一般指的是只能写一次的 ROM;Masked ROM 是掩模型的 ROM,内容出厂时已经写好了,不可再更改,成本低,但批量生产时,若是发现数据错误,只能全部销毁;EEPROM 可以用电子的方式来直接清除及写入数据,许多强调可以升级的主板常用 EEPROM 作为 BIOS 的存储装置。在嵌入式系统中,ROM 的使用相对较少。

在嵌入式系统的存储系统中广泛应用的还有闪存 Flash,兼有 RAM 和 ROM 的优点,既可以在线读写,也可永久存储,一般分为 Nor Flash 和 Nand Flash 两种。

Nand 就是与非(Not AND),里面的单元是按照与非的方式连起来的;Nor 就是 Not OR(或非),里面的单元是按照或非的方式连起来的。

Nand Flash 类似于硬盘,以存储数据为主,又称为 Data Flash,便宜,片容量大,目前主流容量已达 2GB,但是性能不如 Nor。Nor Flash 则类似于内存,以存储程序代码为主,又称为 CodeFlash,CPU 能直接处理,但片容量较低,主流容量为 512MB。

Nand Flash 与 Nor Flash 除了容量不同外,读写速度也不同。

在 Nand Flash 中,存储单元被分成页,由页组成块。根据容量不同,块和页的大小有所不同,而组成块的页的数量也不同,如 8MB 的模块,页大小为(512+16)B、块大小为(8K+256)B;而 2MB 的模块,页大小为(256+8)B、块大小为(4K+128)B。

Nand Flash 存储单元的读写是以块和页为单位来进行的。实际上，Nand Flash 可以看做是顺序读取的设备，它仅用 8 比特的 I/O 端口就可以存取以页为单位的数据。正因为这样，它在写和擦文件特别是连续的大文件时，与 Nor Flash 相比速度快得多。Nand Flash 的不足在于随机读速度较慢，这恰好是 Nor Flash 的优点所在：Nor Flash 的随机读速度较快。正因为这些特点，所以 Nand Flash 适合用在大容量的存储应用中，而 Nor Flash 适合用在程序存储应用中。

为了实现存储系统的访问，ARM 系统设置了存储器控制器和 Nand Flash 控制器。存储器控制器是 ARM 系统最重要的部分，是 ARM 系统与一般 8 位微控制器的最大区别。一般微控制器是靠 I/O 端口来控制对外信号的，但 ARM 系统主要是靠存储器控制器来控制对外信号的，所以 ARM 系统可控制比较复杂的存储，提供访问外接存储器（包括 SDRAM、Nor Flash）所需的存储器控制信号。而 Nand Flash 控制器用于实现 ARM 系统对 Nand Flash 的访问。

下面将分别介绍 S3C2440A 的存储器控制器和 Nand Flash 控制器的特性。

2. S3C2440A 的存储器控制器特性

S3C2440A 的存储器控制器主要具有以下特性。

（1）支持大端、小端模式。

（2）地址空间：总共 4GB，其中 1GB 空间用于连接外部存储器，这 1GB 空间分为 8 块，每块 128MB。

（3）Bank0 支持 16/32 位数据存取，其他 Bank 支持 8/16/32 位数据存取。Bank0～Bank6 的起始地址固定，Bank7 的起始地址可调整。Bank6 和 Bank7 的大小相等，大小可通过编程设定，如表 4-1 所示。

表 4-1　Bank6 和 Bank7 地址及容量大小

容量	2MB	4MB	8MB	16MB	32MB	64MB	128MB
Bank6							
起始地址	0x30000000	0x30000000	0x30000000	0x30000000	0x30000000	0x30000000	0x30000000
末地址	0x301FFFFF	0x303FFFFF	0x307FFFFF	0x30FFFFFF	0x31FFFFFF	0x33FFFFFF	0x37FFFFFF
Bank7							
起始地址	0x30200000	0x30400000	0x30800000	0x31000000	0x32000000	0x34000000	0x38000000
末地址	0x303FFFFF	0x307FFFFF	0x30FFFFFF	0x31FFFFFF	0x33FFFFFF	37FFFFFF	3FFFFFFF

（4）所有存储器块的访问周期可编程，总线访问周期可通过插入外部 WAIT 时间来延长。

（5）支持 SDRAM 的自刷新和掉电模式。

S3C2440A 复位后存储器映射如图 4-4 所示。

0 号存储块可以外接 SRAM 或具有 SRAM 接口特性的 ROM（如 NOR Flash）。当0 号存储块作为 ROM 区，完成引导装入工作时（即从 0x00000000 处启动），0 号存储块的总线宽度应在第一次访问 ROM 数据前根据 OM1、OM0 在复位时的逻辑组合来确定。

3. 存储器控制器寄存器

存储器控制器主要是通过对其特殊功能寄存器的设置来实现相应的功能的，下面将介绍各特殊功能寄存器。

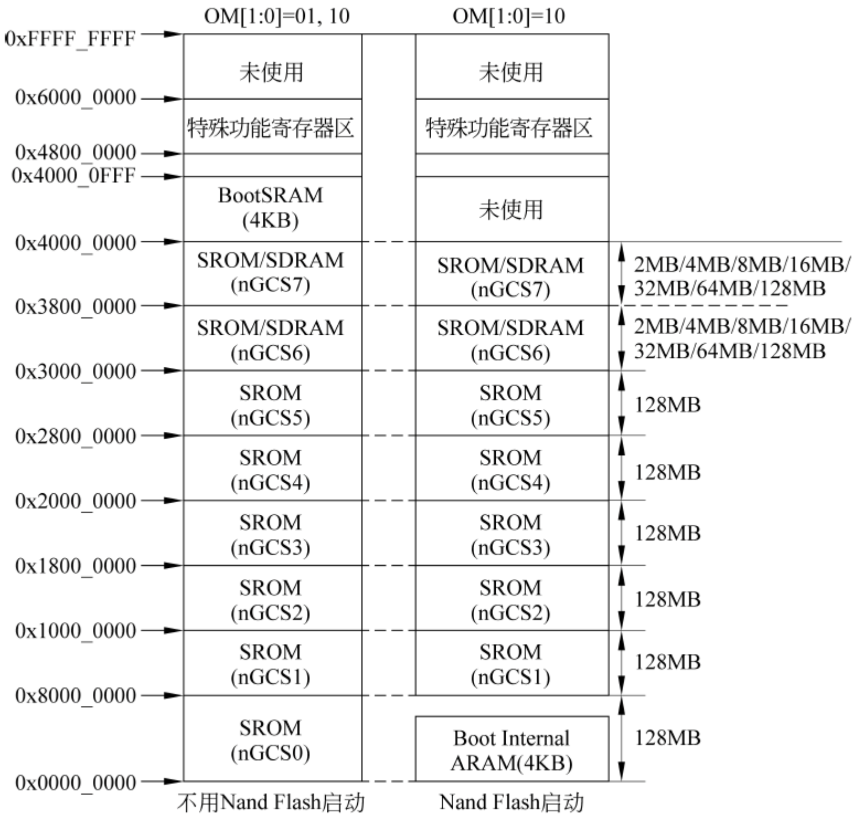


图 4-4　S3C2440A 复位后存储器映射

（1）总线宽度/等待控制寄存器（BWSCON）

总线宽度/等待控制寄存器用于设置各存储块的数据宽度，以及是否使能 nWAIT。地址为 0x48000000，复位时的初始值为 0x00000000。BWSCON 寄存器中各位的具体定义如表 4-2 所示。

表 4-2　BWSCON 寄存器中各位的定义

位	描　述	初始状态
[31]	确定对 7 号存储块是否使用 UB/LB 0=不使用 UB/LB　1=使用 UB/LB	0
[30]	确定对 7 号存储块的等待状态 0=WAIT 无效，1=WAIT 使能	0
[29:28]	确定对 7 号存储块的数据总线宽度 00=8 位，01=16 位，10=32 位，11=reserved	0
[27]	确定对 6 号存储块是否使用 UB/LB 0=不使用 UB/LB　1=使用 UB/LB	0
[26]	确定对 6 号存储块的等待状态 0=WAIT 无效，1=WAIT 使能	0
[25:24]	确定对 6 号存储块的数据总线宽度 00=8 位，01=16 位，10=32 位，11=reserved	0
[23]	确定对 5 号存储块是否使用 UB/LB 0=不使用 UB/LB　1=使用 UB/LB	0

续表

位	描　　述	初始状态
[22]	确定对 5 号存储块的等待状态 0=WAIT 无效,1=WAIT 使能	0
[21:20]	确定对 5 号存储块的数据总线宽度 00=8 位,01=16 位,10=32 位,11=reserved	0
[19]	确定对 4 号存储块是否使用 UB/LB 0=不使用 UB/LB　1=使用 UB/LB	0
[18]	确定对 4 号存储块的等待状态 0=WAIT 无效,1=WAIT 使能	0
[17:16]	确定对 4 号存储块的数据总线宽度 00=8 位,01=16 位,10=32 位,11=reserved	0
[15]	确定对 3 号存储块是否使用 UB/LB 0=不使用 UB/LB　1=使用 UB/LB	0
[14]	确定对 3 号存储块的等待状态 0=WAIT 无效,1=WAIT 使能	0
[13:12]	确定对 3 号存储块的数据总线宽度 00=8 位,01=16 位,10=32 位,11=reserved	0
[11]	确定对 2 号存储块是否使用 UB/LB 0=不使用 UB/LB　1=使用 UB/LB	0
[10]	确定对 2 号存储块的等待状态 0=WAIT 无效,1=WAIT 使能	0
[9:8]	确定对 2 号存储块的数据总线宽度 00=8 位,01=16 位,10=32 位,11=reserved	0
[7]	确定对 1 号存储块是否使用 UB/LB 0=不使用 UB/LB　1=使用 UB/LB	0
[6]	确定对 1 号存储块的等待状态 0=WAIT 无效,1=WAIT 使能	0
[5:4]	确定对 1 号存储块的数据总线宽度 00=8 位,01=16 位,10=32 位,11=reserved	0
[3]	—	—
[2:1]	确定对 0 号存储块数据总线宽度 01=16 位,10=32 位,这个状态也可以通过 OM1、OM0 引脚确定	—
[0]	—	—

(2) 存储块控制寄存器(BANKCON0～BANKCON7)

每个存储块对应一个控制寄存器,BANKCON0～BANKCON5 分别对应 0～5 号存储块,其地址分别是 0x48000004、0x48000008、0x4800000C、0x48000010、0x4800014、0x48000018。复位后的初始值为 0x0700。BANKCON0～BANKCON5 寄存器中各位的具体定义如表 4-3 所示。

BANKCON6、BANKCON7 分别对应 6 号存储块、7 号存储块,地址分别是 0x4800001C、0x48000020。复位后的初始状态为 0x18008。BANKCON6、BANKCON7 寄存器中各位的具体定义如表 4-4 所示。

表 4-3 BANKCON0～BANKCON5 寄存器中各位的定义

BANKCONn	位	描 述	初始状态
Tacs	[14:13]	确定 nGCSn 信号有效前的地址建立时间 00=0 时钟 01=1 时钟 10=2 时钟 11=4 时钟	00
Tcos	[12:11]	确定 nOE 信号有效前的片选建立时间 00=0 时钟 01=1 时钟 10=2 时钟 11=4 时钟	00
Tacc	[10:8]	确定访问周期，注意，当 nWAIT 信号有效时，访问周期≥4 时钟周期 000=1 时钟 001=2 时钟 010=3 时钟 011=4 时钟 100=6 时钟 101=8 时钟 110=10 时钟 111=14 时钟	111
Tcoh	[7:6]	确定 nOE 信号失效后片选信号保持的时间 00=0 时钟 01=1 时钟 10=2 时钟 11=4 时钟	00
Tcah	[5:4]	确定 nGCSn 信号失效后有效地址保持的时间 00=0 时钟 01=1 时钟 10=2 时钟 11=4 时钟	00
Tacp	[3:2]	确定页模式访问周期 00=2 时钟 01=3 时钟 10=4 时钟 11=6 时钟	00
PMC	[1:0]	确定页模式 00=常规(1data) 01=4 data 10=8 data 11=16 data	00

表 4-4 BANKCON6、BANKCON7 寄存器中各位的定义

BANKCONn	位	描 述	初始状态
MT	[16:15]	确定 6 号存储块和 7 号存储块的存储器类型 00=ROM or SRAM 01=FP DRAM 10=EDO DRAM 11=SDRAM	11
当存储器类型为 SRAM 或 ROM 时(MT=00)，需用下面 15 位			
Tacs	[14:13]	确定 nGCSn 信号有效前的地址建立时间 00=0 时钟 01=1 时钟 10=2 时钟 11=4 时钟	00
Tcos	[12:11]	确定 nOE 信号有效前的片选建立时间 00=0 时钟 01=1 时钟 10=2 时钟 11=4 时钟	00
Tacc	[10:8]	确定访问周期，注意，当 nWAIT 信号有效时，访问周期≥4 时钟周期 000=1 时钟 001=2 时钟 010=3 时钟 011=4 时钟 100=6 时钟 101=8 时钟 110=10 时钟 111=14 时钟	111
Tcoh	[7:6]	确定 nOE 信号失效后片选信号保持的时间 00=0 时钟 01=1 时钟 10=2 时钟 11=4 时钟	00
Tcah	[5:4]	确定 nGCSn 信号失效后有效地址保持的时间 00=0 时钟 01=1 时钟 10=2 时钟 11=4 时钟	00
Tacp	[3:2]	确定页模式访问周期 00=2 时钟 01=3 时钟 10=4 时钟 11=6 时钟	00
PMC	[1:0]	确定页模式 00=常规(1data) 01=4 data 10=8 data 11=16 data	00
当存储器类型为 SDRAM(MT=11)时，需用下面 4 位			
Trcd	[3:2]	确定 RAS 对 CAS 的延时 00=2 时钟 01=3 时钟 10=4 时钟 11=6 时钟	10
SCAN	[1:0]	确定列地址位数 00=8 位 01=9 位 10=10 位	00

(3) 刷新控制寄存器(REFRESH)

DRAM/SDRAM 类型存储器需要使用刷新控制寄存器。其地址是 0x48000024,复位后的初始状态为 0xac0000。REFRESH 寄存器各位的定义如表 4-5 所示。

表 4-5　REFRESH 寄存器各位的定义

REFRESH	位	描　　述	初始状态
REFEN	[23]	确定 DRAM/SDRAM 是否启用刷新功能 0=无效　1=有效(自刷新或自动刷新)	1
TREFMD	[22]	确定刷新模式 0=Auto 模式(自动刷新)　1=Self 模式(自刷新) 在自刷新时,DRAM/SDRAM 控制线需要适当的电平驱动	0
Trp	[21:20]	确定 RAS 有效建立的时间 DRAM 00=1.5 时钟　01=2.5 时钟　10=3.5 时钟　11=4.5 时钟 SDRAM 00=2 时钟　01=3 时钟　10=4 时钟　11=不支持	10
Tsrc	[19:18]	确定 SDRAM RAS 预充电时间 00=4 时钟　01=5 时钟　10=6 时钟　11=7 时钟	11
Tchr	[17:16]	确定 CAS 保持时间(DRAM) 00=1 时钟　01=2 时钟　10=3 时钟　11=4 时钟 SDRAM 刷新周期:Trc=Tsrc+Trp	00
Reserved	[15:11]	没有用到	00000
Refresh Counter	[10:0]	确定 DRAM/SDRAM 刷新计数值 刷新周期=(2^{11}-刷新计数值+1)/HCLK 例如,如果刷新周期是 15.6μs,并且 HCLK=60MHz,那么刷新计数值=2^{11}+1-60×15.6=1113	0x0

(4) 存储块大小控制寄存器(BANKSIZE)

BANKSIZE 寄存器的主要功能是确定 6 号存储块和 7 号存储块的容量大小。其地址是 0x48000028,复位后的初始状态为 0x02。BANKSIZE 寄存器各位的定义如表 4-6 所示。

表 4-6　BANKSIZE 寄存器各位的定义

位	描　　述	初始状态
[7]	确定是否启用 ARM 内核突发操作 0=无效突发操作　1=启用突发操作	0
[6]	没有用到	0
[5]	确定 SDRAM 是否启用省电模式 0=不启用　1=启用	0
[4]	SCLK 仅在 SDRAM 访问周期内为减少电源消耗时被激活。当 SDRAM 没有被访问时,SCLK 变成低电平 SCLK=0 时总是激活 SCLK=1 时仅在访问周期才被激活(推荐)	1
[3]	没有用到	0
[2:0]	确定 6 号存储块/7 号存储块的容量 010=128MB　001=64MB　000=32MB　111=16MB 110=8MB　101=4MB　100=2MB	010

4. SDRAM 应用编程

S3C2440A 芯片本身提供了与 SDRAM 进行直接接口的解决方案,因此,不需要通过编程来实现它们所需的接口时序,而只需对与存储器控制器相关的寄存器进行适当配置。这个配置工作一般在启动代码中完成,系统在每次上电后但还未开始执行 C 语言程序之前,先配置好 SDRAM 的特性参数,然后开始执行 C 程序。

若 S3C2440A 系统主频 HCLK = 66MHz,连接的 SDRAM 采用两片 16 位 HY57V561620,HY57V561620 中的每个 bank 为 256 列,即 8 位列地址线;由于需要刷新功能,REFEN=1;刷新模式选择刷新时间和结果可预期的自动刷新模式,TREFMD=0;Trp 是从 bank 预充电到下一次行激活的时间间隔。通过查阅 HY57V561620 数据手册,取 Trp=00,即两个时钟;自动刷新操作指令要求在每 64ms 中至少执行 4096 次,因此套用表格中的公式,刷新技术值为 1049;同样从数据手册中可以得到每次刷新所需要的时间大约为 5 个时钟,取 Trc=01。Tchr 主要针对 DRAM 设置,这里不必考虑。

将存储器的设置值都放在特定设置文件中,这种设计使得读者在更改存储器的编号时不需要更改程序,只需更改其设置文件即可。部分程序如下(因 bank1～bank5 参数设置相同、bank6～bank7 参数设置相同,故限于篇幅只给出 bank0、bank1、bank6 及 REFRESH 参数设置值):

```
;bank0 参数
B0_Tacs      EQU     0x3            ;0clk
B0_Tcos      EQU     0x3            ;0clk
B0_Tacc      EQU     0x7            ;14clk
B0_Tcoh      EQU     0x3            ;0clk
B0_Tah       EQU     0x3            ;0clk
B0_Tacp      EQU     0x1
B0_PMC       EQU     0x0            ;正常模式
B1_Tacs      EQU     0x1            ;0clk
B1_Tcos      EQU     0x1            ;0clk
B1_Tacc      EQU     0x6            ;14clk
B1_Tcoh      EQU     0x1            ;0clk
B1_Tah       EQU     0x1            ;0clk
B1_Tacp      EQU     0x0            ;2clk
B1_PMC       EQU     0x0            ;normal
;bank6 参数
B6_MT        EQU     0x3            ;SDRAM
B6_Trcd      EQU     0x1            ;3clk
B6_SCAN      EQU     0x1            ;9 位
;REFRESH 参数
REFEN        EQU     0x1            ;启用刷新功能
TREFMD       EQU     0x0            ;CBR(CAS before RAS)/Auto refresh
Trp          EQU     0x1            ;3clk
Tsrc         EQU     0x1            ;5clkTrc=Trp(3)+Tsrc(5)=8clock
Tchr         EQU     0x2            ;3clk
REFCNT       EQU     1268           ;HCLK=100MHz,(2048+1-7.81*100)
;寄存器存储地址定义
BWSCON       EQU     0x48000000     ;总线宽度和 WAIT 控制寄存器
BANKCON0     EQU     0x48000004     ;bank0 控制寄存器
```

```
BANKCON1      EQU      0x48000008        ;bank1 控制寄存器
BANKCON6      EQU      0x4800001c        ;bank6 控制寄存器
REFRESH       EQU      0x48000024        ;DRAM/SDRAM 刷新寄存器
BANKSIZE      EQU      0x48000028        ;存储块大小控制寄存器
MRSRB6        EQU      0x4800002c        ;bank6 模式设置寄存器
SMRDATA       DATA                       ;定义初始化数据区
DCD  (0+(1<<4)+(13<<8)+(13<<12)+(5<<16)+(1<<20)+(1<<24)+(1<<28))    ;BWSCON
DCD  ((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tacp<<2)+(B0
_PMC))        ;GCS0
DCD  ((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Tacp<<2)+(B1
_PMC))        ;GCS1
DCD  ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))        ;GCS6
DCD  ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Tsrc<<18)+(Tchr<<16)+REFCNT)    ;REFRESH
DCD 0x32                                 ;SCLK 省电模式,容量 128MB/128MB
DCD 0x30                                 ;MRSR6 CL=3clk
/**************************************************************************
* Description: 使用汇编语言初始化存储器控制器寄存器
**************************************************************************/
;Set memory control registers
    ldr       r0,=SMRDATA          ;取初始化数据所在地址
    ldr       r1,=BWSCON           ;取 BWSCON 地址
    add       r2,r0,#52            ;SMRDATA 处共 13 个初始化数据对应 13 个存储器寄存器
0
    ldr       r3,[r0],#4
    str       r3,[r1],#4
    cmp       r2,r0                ;判断是否完成初始化
    bne       %B0
    mov       r1,#256
0
    subs      r1,r1,#1             ;延时,等待自刷新完成
    bne       %B0
```

4.3.2 Nand Flash 控制器

任务：掌握 S3C2440A 的 Nand Flash 控制器的功能、结构、初始化及操作。

Nor Flash 和 Nand Flash 是目前市场上两种主要的非易失性闪存，可以用来固化嵌入式系统中的程序。

Nor Flash 存储器的容量较小、写入速度较慢，但因其随机读取速度快，在嵌入式系统中常用于存储程序，程序能够在 Nor Flash 上直接运行。但 Nor Flash 价格较高，而 SDRAM 和 Nand Flash 存储器相对经济，这就促使一些用户希望在 Nand Flash 上启动和引导系统，在 SDRAM 上执行主程序。

由于 Nand Flash 具有特殊的结构和寻址方式（串行），因而在 Nand Flash 上无法执行代码。因此，若要执行存储在 Nand Flash 上的程序，需先将代码复制到 SDRAM 中，然后在 SDRAM 中执行。为了支持 Nand Flash 的系统引导，S3C2440A 配备了一个名为 Steppingstone 的内部 SRAM 型缓冲器。当系统启动时，Nand Flash 上的前 4KB 作为引导代码将被自动载入到 Steppingstone 中，然后系统自动执行这些引导代码。在一般情况下，

这 4KB 的引导代码会将 Nand Flash 上的内容复制到 SDRAM 中。复制完成后，主程序将在 SDRAM 上执行。

使用 S3C2440A 内部硬件的 ECC(Errer Checking and Correcting，错误检查和纠正)功能可以对 Nand Flash 上的数据进行有效性的检查。

1. Nand Flash 控制器特性及内部结构

(1) 两种工作模式：软件模式(Nand Flash 模式)和自动导入模式。

软件模式：用户可以直接访问 Nand Flash，如对 NADN Flash 的读、擦除和编程。

自动导入模式：复位后，若启用自动导入模式，则 Nand Flash 的前 4KB 引导代码被自动传输到 4KB 的 Steppingstone 中，Steppingstone 被映射到存储块 bank0(nGCS0)。然后 CPU 在 Steppingstone 中开始执行引导代码。在自动导入模式下，不进行 ECC 检测。因此，Nand Flash 的前 4KB 应确保不能有位错误(一般 Nand Flash 厂家都确保)。

(2) Nand Flash 存储器接口：支持 256 字、512 字节、1K 字节、2K 字节的页。

(3) 接口：8/16 位的 Nand Flash 存储器接口总线。

(4) 硬件 ECC 生成器：生成 ECC 码，用于进行错误检查和纠正。

(5) SFR 接口：支持小端模式，实现对数据和 ECC 数据寄存器的字节、半字、字访问，对其他寄存器的字访问。

(6) Steppingstone 接口：支持大端模式和小端模式，可实现对字节、半字和字的访问。

(7) Steppingstone 缓冲器可以在 Nand Flash 启动后用于其他目的。

2. Nand Flash 控制器结构

在许多嵌入式系统中均设计有 Nand Flash 存储器，作为系统辅助存储器，但 Nand Flash 存储器与微处理器之间的接口较为复杂，存取数据通常采用串行的 I/O 方式。并且，Nand Flash 缺乏统一的接口规范，这增加了设计接口的难度。

在 S3C2440 内部集成了 Nand Flash 控制器，内部结构(16 位数据总线)如图 4-5 所示。

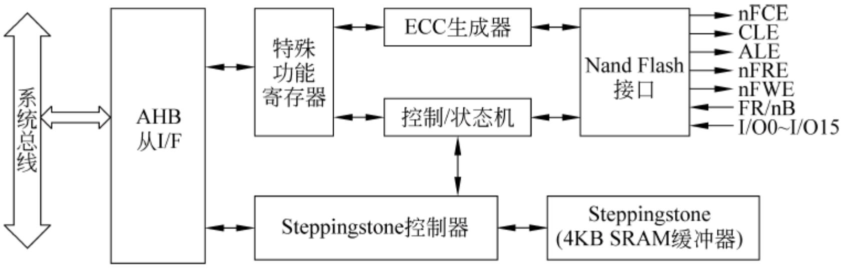


图 4-5 Nand Flash 控制器内部结构

(1) ECC 生成器

Nand Flash 控制器包括 4 个 ECC 模块。其中，两个模块(一个用于 data[7:0]，一个用于 data[15:8])可以用于(上限)2048B 的 ECC 奇偶校验码的生成，另外两个模块(一个用于 data[7:0]，一个用于 data[15:8])可以用于(上限)16B 的 ECC 奇偶校验码的生成。

(2) Steppingstone 缓冲器及控制器

OM[1:0]=00 时，系统采用 Nand Flash 存储器启动。为了支持 Nand Flash 的引导，S3C2440A 配备了一个内部的 SRAM 缓冲器，名为 Steppingstone。启动时，Nand Flash 上的前 4KB 将通过控制器装载到 Steppingstone 内且装载到 Steppingstone 中的启动代码会被执行。在 Nand Flash 启动后，Steppingstone 缓冲器可以被用于其他用途。

(3) 引脚配置

① I/O[7:0]：数据/命令/地址的输入/输出口，与低 8 位数据总线相连，分时传送命令、地址和数据；对于 16 位 Nand Flash 接口还有 I/O[15:8]。

② CLE：命令输入开启信号，输出。高电平有效，表明写入的是命令字。

③ ALE：地址输入开启信号，输出。高电平有效，表明写入的是地址。

④ nFCE：Nand Flash 片选信号，输出。低电平有效，开启芯片。

⑤ nFRE：Nand Flash 读允许信号，输出。低电平有效，Flash 会根据写入的命令和地址从 I/O 口输出数据。

⑥ nFWE：Nand Flash 写允许信号，输出。低电平有效，向 Flash 写入数据。

⑦ FR/nB：Nand Flash 准备好/忙，输入，低电平表示设备忙。

除了以上各个接口信号外，S3C2440A 还提供了 NCON、GPG13-GPG15 等引脚与 Nand Flash 进行连接，详见 Nand Flash 配置寄存器的定义。

3. NADN Flash 寄存器

(1) Nand Flash 配置寄存器(NFCONF)

NFCONF 可读写，端口地址为 0x4E000000，初始值为 0x0000100X(X 指不确定)，各位的定义如表 4-7 所示。其中，后 4 位 AdvFlash、PageSize、AddrCycle、BusWidth 分别对应微处理器的引脚 NCON0、GPG13、GPG14、GPG15。

表 4-7　Nand Flash 配置寄存器的定义

NFCONF	位	功能描述
保留	[24]	保留
TACLS	[13:12]	确定 CLE 和 ALE 持续的时间　持续时间＝HCLK × TACLS
保留	[11]	保留
TWRPH0	[10:8]	确定 TWRPH0 持续的时间　持续时间＝HCLK×(TWRPH0＋1)
保留	[7]	保留
TWRPH1	[6:4]	确定 TWRPH1 持续的时间　持续时间＝HCLK×(TWRPH1＋1)
AdvFlash(只读)	[3]	用于 Nand Flash 存储器选择 0：支持 256 字/512 字节页大小的普通 Nand Flash 1：支持 1K 字/2K 字节页大小的高级 Nand Flash 该位由复位或从睡眠模式唤醒后的 NCON 引脚状态确定
PageSize(只读)	[2]	用于 Nand Flash 存储器页容量选择，与 NCON(即 AdvFlash)有关 0：页＝256 字(NCON＝0)或页＝1K 字(NCON＝1) 1：页＝512 字节(NCON＝0)或页＝2K 字节(NCON＝1) 该位由复位或从睡眠模式唤醒后的 GPG13 引脚状态确定。复位后，GPG13 可用于通用输入/输出端口或外部中断
AddrCycle(只读)	[1]	用于 Nand Flash 存储器地址周期选择，与 NCON(即 AdvFlash)有关 0：3 个地址周期(NCON＝0)或 4 个地址周期(NCON＝1) 1：4 个地址周期(NCON＝0)或 5 个地址周期(NCON＝1) 该位由复位或从睡眠模式唤醒后的 GPG14 引脚状态确定。复位后，GPG14 可用于通用输入/输出端口或外部中断
BusWidth(读写)	[0]	用于 Nand Flash 存储器总线宽度选择 0：8 位　　　1：16 位该位由复位或从睡眠模式唤醒后的 GPG15 引脚状态确定。复位后，GPG15 可用于通用输入/输出端口或外部中断

(2) Nand Flash 控制寄存器(NFCONT)

NFCONT 可读写,端口地址为 0x4E000004,初始值为 0x1062,各位的定义如表 4-8 所示。

表 4-8 Nand Flash 控制寄存器各位的定义

NFCONT	位	功能描述
保留	[15:14]	保留
lock-tight	[13]	紧锁定设置 0:禁止锁定　　1:允许锁定 该位一旦设置为 1 就不能清除。只有复位或睡眠唤醒才能清除(不能用软件清除) 为 1 时,在 NFSBLK(0x4E000038)至 NFEBLK(0x4E00003C)－1 中设置的区域是锁定的,不能写或擦除,只能读
soft lock	[12]	软锁定设置 0:禁止锁定　　1:允许锁定 软锁定域可以在任何时刻通过软件更改。为 1 时,在 NFSBLK(0x4E000038)至 NFEBLK(0x4E00003C)－1 中设置的区域是锁定的,不能写或擦除,只能读
保留	[11]	保留
EnbIllegalAccINT	[10]	非法访问中断控制 0:禁止中断　　1:允许中断 CPU 试图改写或擦除锁定区域(在 NFSBLK(0x4E000038)至 NFEBLK(0x4E00003C)－1 中设置的区域)时发生非法访问中断
EnbRnBINT	[9]	RnB 状态输入信号跳变中断控制 0:禁止 RnB 中断　　1:允许 RnB 中断
RnB_TransMode	[8]	RnB 跳变检测设置 0:检测上升沿　　1:检测下降沿
保留	[7]	保留
SpareECCLock	[6]	锁定空闲区 ECC 奇偶校验码 0:不锁定　　1:锁定
MainECCLock	[5]	锁定主数据区 ECC 奇偶校验码 0:不锁定　　1:锁定
InitECC	[4]	确定 ECC 编码器/解码器初始化 0:不初始化 ECC　　1:初始化 ECC
保留	[3:2]	保留
Reg_nCE	[1]	Nand Flash 存储器 nFCE 信号控制 0:强制为低电平　　1:强制为高电平 注意:启动时,该位自动控制。该值只有在 MODE 位为 1 时才有效
MODE	[0]	确定 Nand Flash 控制器的操作模式 0:禁止　　1:允许

(3) Nand Flash 命令寄存器(NFCMMD)

NFCMMD 可读写,端口地址为 0x4E000008,初始值为 0x00,各位的定义如表 4-9 所示。

(4) Nand Flash 地址寄存器(NFADDR)

NFADDR 可读写,端口地址为 0x4E00000C,初始值为 0x0000XX00,各位的定义如表 4-10 所示。

表 4-9　Nand Flash 命令寄存器各位的定义

NFCMMD	位	功能描述	NFCMMD	位	功能描述
保留	[15:8]	保留	NFCMMD	[7:0]	Nand Flash 命令

表 4-10　Nand Flash 地址寄存器各位的定义

NFADDR	位	功能描述	NFADDR	位	功能描述
保留	[15:8]	保留	NFADDR	[7:0]	Nand Flash 存储地址

(5) Nand Flash 数据寄存器(NFDATA)

NFDATA 可读写,端口地址为 0x4E000010,初始值为 0x00,各位的定义如表 4-11 所示。

表 4-11　Nand Flash 数据寄存器各位的定义

NFDATA	位	功 能 描 述
NFDATA	[31:0]	Nand Flash 输入/输出的数据

NFDATA 的配置相对复杂,不但要区分所接 Nand Flash 存储器的 I/O 总线宽度,还要区分大、小端模式以及数据操作的类型(即字、字节、半字)。总的规则是:不能一次性传输的数据需要按照大/小端模式从最高/低字节开始依次传输,分次传完。例如,在小端模式下要利用 8 位 I/O 总线传输 32 位数据 0x89a5,则需要利用 I/O[7:0]先后依次传输 5、10、9、8,分 4 个周期完成。

(6) Nand Flash 主数据区域 ECC 寄存器(NFMECCD0/1)

NFMECCD0/1 可读写,端口地址为 0x4E000014 和 0x4E000018,初始值为 0x00,NFMECCD0 各位的定义如表 4-12 所示。

表 4-12　Nand Flash 主数据区域 ECC 寄存器 0 各位的定义

NFMECCD0	位	功 能 描 述
ECCData1_1	[31:24]	第 2 个 I/O[15:8]的 ECC
ECCData1_0	[23:16]	第 2 个 I/O[7:0]的 ECC 注意:在软件模式下,需要读 Nand Flash 存储器的第 2 个 ECC 值时就读 ECCData1
ECCData0_1	[15:8]	第 1 个 I/O[15:8]的 ECC
ECCData0_0	[7:0]	第 1 个 I/O[7:0]的 ECC 注意:在软件模式下,需要读 Nand Flash 存储器的第 1 个 ECC 值时就读 ECCData0 该寄存器的读取与 NFDATA 的读取相同

NFMECCD1 各位的定义如表 4-13 所示。

在软件模式下,ECC 模块为所有的读写数据生成 ECC 奇偶校验码,因此通过将 InitECC 位(NFCONT[4])置 1 来重置 ECC 值,且在读写数据之前将 MainECCLock (NFCONT[5])位清 0。不管读还是写数据,ECC 模块都在寄存器 NFMECC0/1 上生成 ECC 奇偶校验码。在读写一页后(不包括空闲区域数据),MainECCLock=1,ECC 奇偶校验码被锁定,且 ECC 状态寄存器的值不会被改变。

表 4-13　Nand Flash 主数据区域 ECC 寄存器 1 各位的定义

NFMECCD1	位	功能描述
ECCData3_1	[31:24]	第 4 个 I/O[15:8]的 ECC
ECCData3_0	[23:16]	第 4 个 I/O[7:0]的 ECC 注意：在软件模式下，需要读 Nand Flash 存储器的第 4 个ECC 值时就读 ECCData3
ECCData2_1	[15:8]	第 3 个 ECC 的 I/O[15:8]
ECCData2_0	[7:0]	第 3 个 ECC 的 I/O[7:0] 注意：在软件模式下，需要读 Nand Flash 存储器的第 3 个ECC 值时就读 ECCData2 该寄存器的读取与 NFDATA 的读取相同

(7) Nand Flash 空闲区域 ECC 寄存器(NFSECCD)

NFSECCD 可读写，端口地址为 0x4E00001C，初始值为 0x00，各位的定义如表 4-14 所示。

表 4-14　Nand Flash 空闲区域 ECC 寄存器各位的定义

NFSECCD	位	功能描述
ECCData1_1	[31:24]	第 2 个 ECC 的 I/O[15:8]
ECCData1_0	[23:16]	第 2 个 ECC 的 I/O[7:0] 注意：在软件模式下，需要读 Nand Flash 存储器的第 2 个ECC 值时就读该寄存器
ECCData0_1	[15:8]	第 1 个 ECC 的 I/O[15:8]
ECCData0_0	[7:0]	第 1 个 ECC 的 I/O[7:0] 注意：在软件模式下，需要读 Nand Flash 存储器的第 1 个 ECC 值时就读该寄存器 该寄存器的读取与 NFDATA 的读取相同

生成空闲区域 ECC 奇偶校验码前，将 SpareECCLock 位(NFCONT[6])清 0。读或写数据时，空闲区域 ECC 模块在寄存器 NFSECC 上生成 ECC 奇偶校验码。在读写空闲区域后，SpareECCLock=1，ECC 奇偶校验码被锁定且 ECC 状态寄存器的值不会被改变。

可以将生成的 ECC 奇偶校验码记录到空闲区域，也可以进行位错误检查。

注意　NFSECCD 为空闲区域的 ECC 服务(通常将主数据区域的 ECC 值写到空闲区域，这些值和 NFMECC0/1 中的值一样)，且从主数据区域中生成。

(8) Nand Flash 操作状态寄存器(NFSTAT)

NFSTAT 可读写，端口地址为 0x4E000020，初始值为 0xX3，各位的定义如表 4-15 所示。

(9) Nand Flash ECC0/1 状态寄存器(NFESTAT0/1)

NFESTAT0/1 均可读写，端口地址为 0x4E000024 和 0x4E000028，分别作为 I/O[7:0] 和 I/O[15:8]的 ECC 状态寄存器，初始值为 0x00，各位的定义如表 4-16 所示，限于篇幅，这里只给出 NFESTAT0 针对 I/O[15:8]的 ECC 状态，NFESTAT1 与之类似。

表 4-15　Nand Flash 操作状态寄存器各位的定义

NFSTAT	位	功 能 描 述
保留	[7:4]	保留
IllegalAccess	[3]	一旦 Soft Lock 或 Lock-tight 允许，对存储器的非法访问将使得该位被置为 1 0：未检测到非法访问 1：检测到非法访问
RnB_TransDetect	[2]	RnB 从低电平跳变为高电平时，该位被设置并在允许的情况下发生中断。向该位写 1 可实现位清除 0：未检测到 RnB 跳变 1：检测到 RnB 跳变跳变 检测在 RnB_TransMode(NFCONT[8])中设置
nCE(只读)	[1]	nCE 输出引脚的状态
RnB(只读)	[0]	RnB 输入引脚的状态 0：存储器忙 1：存储器准备好

表 4-16　Nand Flash NFESTAT0 状态寄存器各位的定义

NFESTAT0	位	功 能 描 述
SErrorDataNo	[24:21]	指明在空闲区哪一个序号的数据错
SErrorBitNo	[20:18]	指明在空闲区哪一位错
MErrorDataNo	[17:7]	指明在主数据区哪一个序号的数据错
MErrorBitNo	[6:4]	指明在主数据区哪一位错
SpareError	[3:2]	表示空闲区域是否有位错误发生 00：无　01：1 位错(可检测的) 10：多位错　11：ECC 区错
MainError	[1:0]	表示主数据区域是否有位错误发生 00：无　01：1 位错(可检测的) 10：多位错　11：ECC 区错

(10) Nand Flash 主数据区域 ECC 状态寄存器(NFMECC0/1)

NFMECC0/1 只读，端口地址为 0x4E00002C 和 0x4E000030，分别对应 data[7:0]和 data[15:8]的 ECC 寄存器，初始值不确定。若 MainECCLock(NFCONT[5])为 0(不锁定)，Nand Flash 控制器会在读或写空闲区数据时生成 NFMECC0/1。各位的定义如表 4-17 所示，为节约篇幅，此处只给出对应 data[7:0]的 NFMECC0，NFMECC1 与之类似，对应 data[15:8]。

表 4-17　Nand Flash 主数据区域 ECC 状态寄存器各位的定义

NFMECC	位	功能描述	NFMECC	位	功能描述
MECC0_3	[31:24]	data[7:0]的 ECC3	MECC0_1	[15:8]	data[7:0]的 ECC1
MECC0_2	[23:16]	data[7:0]的 ECC2	MECC0_0	[7:0]	data[7:0]的 ECC0

(11) Nand Flash 空闲区域 ECC 状态寄存器(NFSECC)

NFSECC 只读，端口地址为 0x4E000034，初始值不确定，对应 I/O[15:0]的 ECC 寄存

器,各位的定义如表 4-18 所示。若 SpareECCLock(NFCONT[6])为 0(不锁定),Nand Flash 控制器会在读或写空闲区数据时生成 NFECC。

表 4-18　Nand Flash 空闲区域 ECC 状态寄存器各位的定义

NFMECC	位	功能描述
SECC1_1	[31:24]	空闲区 I/O[15:8]的 ECC1 状态
SECC1_0	[23:16]	空闲区 I/O[15:8]的 ECC0 状态
SECC0_1	[15:8]	空闲区 I/O[7:0]的 ECC1 状态
SECC0_0	[7:0]	空闲区 I/O[7:0]的 ECC0 状态

(12) Nand Flash 块地址寄存器(NFSBLK 和 NFEBLK)

NFSBLK 和 NFEBLK 可读写,端口地址分别为 0x4E000038 和 0x4E00003C,给出 Nand Flash 编程的首地址和末地址,初始值为 0x00,NFSBLK 各位的定义如表 4-19 所示,NFEBLK 与之类似。

表 4-19　Nand Flash 块地址寄存器各位的定义

NFSBLK	位	功能描述
SBLK_ADDR2	[23:16]	块擦除操作的第 3 个块地址
SBLK_ADDR1	[15:8]	块擦除操作的第 2 个块地址
SBLK_ADDR0	[7:0]	块擦除操作的第 1 个块地址(只有[7:5]有效)

若 lock-tight 或 soft lock(NFCONT[13:12])为 1,NFSBLK 首地址和末地址 NFEBLK 相同,则整个 Nand Flash 锁定,只能读,擦除或写入操作将被视为非法。

4. Nand Flash 的操作

对 Nand Flash 的操作包括 Nand Flash 的擦除和烧写。对 Nand Flash 的编程与擦除是与具体的器件型号紧密相关的,不同器件型号的操作方式是不相同的,所以在操作中要仔细阅读它的相关文档。由于不同厂商的 Nand Flash 在操作命令上可能会有一些细微的差别,Nand Flash 的烧写、擦除程序一般不具有通用性,针对不同厂商、不同型号的 Flash 存储器,程序应做相应的修改。这里以 K9F6408 为例,简单介绍 Nand Flash 的操作,如表 4-20 所示。

表 4-20　K9F6408 系列 Nand Flash 命令设置

功　能	第一总线周期	第二总线周期	器件忙时可接受的命令(用 0 表示)
读第 1、2 区	00h/01h	—	
读第 3 区	50h	—	
读 ID	90h	—	
复位	ffh	—	0
页写入	80h	10h	
块擦除	60h	d0h	
读状态	70h	—	0

K9F6408 的容量为 64MB(8M×8B),存储空间分为 1024 个块,按每块 16 页(行)、每页 528 个字节(列)的组成方式构成。每页 528 个字节的前 512 个字节为主数据存储器,存放用户数据,分两个区:0~255B 为第 1 区,256~511B 为第 2 区;后 16 个字节为辅助数据存

储器，存放 ECC 代码、坏块信息和文件系统代码等，为第 3 区。该芯片内部还有一个容量为 528B 的静态寄存器，称为页寄存器，用来在进行数据存取时作为缓冲区。编程数据和读取的数据可以在寄存器和存储阵列中按 528B 的顺序递增访问。当对芯片的某一页进行读写时，其数据首先被转移到该寄存器中，通过该寄存器和其他芯片进行数据交换，片内的读写操作由片内的处理器自动完成。

K9F6408 的读和写都以页为单位，擦除则以块为单位。这种块—页结构恰好能满足文件系统中划分簇和扇区的结构要求。

K9F6408 的地址通过 8 位端口传送，有效地减少了引脚的数量，并能够保持不同密度器件引脚的一致性，系统可以在不对电路进行改动的情况下升级为高容量的存储器件。

K9F6408 引脚包括 I/O[7:0]（数据输入\输出端口）、CLE（命令锁存使能）、ALE（地址锁存使能）、nCE（片选）、nRE（读使能）、nWE（写使能）、nWP（写保护）、nSE（选择空闲区使能）、R/nB（状态输出）、Vcc（电源）、Vss（接地）、N. C（无连接）。其中，通过 CLE 和 ALE 信号线实现 I/O 口上指令和地址的复用。指令、地址和数据都通过拉低 WE 和 CE 从 I/O 口写入器件中。有些指令只需要一个总线周期完成，例如，复位指令、读指令和状态读指令等；另外一些指令，例如，页写入和块擦除，则需要两个周期，其中一个周期用来启动，另一个周期用来执行。

对 K9F6408 进行编程操作，首先需要通过数据线向 K9F6408 芯片输出“写入命令 0x80”，然后通过数据线输出需要编程的 Nand Flash 存储单元的地址（地址是多字节的，需要顺序输出），接着再输出需要写入的数据，以及输出“命令 0x10”。上述内容输出完成后，读取状态寄存器或 R/nB 引脚，判断 K9F6408 芯片是否忙。若忙，则等待；若不忙，则进行校验。进行校验时，先通过数据线向 K9F6408 芯片输出“写入命令 0x00”，然后通过数据线输出需要校验的 Nand Flash 存储单元的地址，然后读出数据进行比较。若正确，编程工作完成。

4.3.3　中断控制器

任务：掌握 S3C2440A 的中断控制器的功能、结构、初始化及对中断的管理与实现。

1. 中断机制概述

在系统中，当处理器与外围设备交换信息时，若采用查询的方式，则处理器就要浪费很多时间去等待外围设备，这样就存在一个高速的 CPU 与低速的外围设备之间的矛盾。为了解决这个问题，一方面要提高外围设备的工作速率，另一方面发展中断机制。

外围设备在做好进行一次数据输入输出的准备后，产生一个信号传送给微处理器请求传输数据，这个信号称为中断请求；引起中断的原因，或者中断请求信号的来源称为中断源。若微处理器可以进行数据传输，则响应中断请求信号，停止当前正在执行的程序，而转向对该外围设备进行新的输入输出操作，称为中断响应；对外围设备进行新的输入输出操作所执行的程序称为中断服务程序；中断服务程序在内存中的存储地址称为中断向量；处理完中断服务程序后返回原来执行的程序继续执行，称为中断返回。

中断控制器的角色就是响应来自 FIQ（快速中断请求）或 IRQ（普通中断请求）的中断，并请求内核对中断进行处理。当有多个中断同时发生时，中断控制器要决定首先处理哪个

中断。

2. 中断系统的功能

为了实现各种中断请求,中断系统应具有以下功能。

(1) 实现中断及返回。当某一中断源发出中断请求时,CPU 能决定是否响应这个中断请求。若允许响应这个请求,则 CPU 必须在运行的指令执行完后,把断点处的各个寄存器的内容推入堆栈,保留现场;然后 CPU 转到需要处理的中断服务程序的入口,同时清除中断请求触发器;当中断处理完后,再恢复被保留的各个寄存器的值,使 CPU 返回断点,继续执行主程序。

(2) 能够实现优先级排队。通常,系统中有多个中断源,会出现两个或多个中断源同时提出中断请求的情况,这就要求设计者必须事先根据轻重缓急,给每个中断源确定一个中断优先级。当多个中断源同时发出中断请求时,CPU 能够找到优先级最高的中断源,并响应它的中断请求;在高优先级的中断源处理完后,再响应优先级较低的中断源。

(3) 高级中断源能够中断低级的中断处理。当 CPU 进行某一中断处理时,若用优先级更高的中断源发出中断申请,则 CPU 要能够中断正在执行的中断服务程序,保留该程序的现场,响应高级中断;在高级中断处理完后,再继续执行被中断的中断服务程序,这一过程称为中断嵌套。

3. 中断方式控制的输入输出操作

典型的中断方式控制的输入输出操作步骤包括以下几个。

(1) 开放中断。

(2) I/O 端口或部件做好准备后发出中断请求。

(3) 中断请求信号有效时,若微处理器允许中断,则保存当前状态,停止现行操作并识别中断源。

(4) 识别最高优先级的中断源,微处理器转向中断服务程序,并应答中断;I/O 端口或部件收到信号,撤销中断请求。

(5) 在中断服务程序中完成数据读写,结束后返回原来执行的程序继续执行。

4. 中断源

在嵌入式系统中需要识别的中断源有许多,如 S3C2410 有 56 个中断源,S3C2440 有 60 个。而通常微处理器能够提供的中断请求信号线是有限的,因此当有中断产生时,需要识别中断源以确定中断向量,找到对应的中断服务程序。

目前,中断源的识别方法主要是向量识别方法:固定中断向量和可变中断向量。

(1) 固定中断向量:各个中断源对应的中断向量固定,各个中断源有各自的中断类型码,根据中断类型码得出中断向量。

(2) 可变中断向量:各个中断源对应的中断向量可设定。

S3C2440A 中的中断控制器协助 ARM9 实现中断管理,采用固定向量识别方式识别中断,可以从 60 个中断源接收中断请求,这些中断源由 S3C2440A 内部外围设备或外部中断请求引脚提供,如 DMA 控制器、UART、IIC 等,这些中断请求经过仲裁后,由中断控制器通过 ARM920T 的 FIQ 或 IRQ 请求中断,对应的 ARM920T 有两个类型的中断模式:FIQ 和 IRQ。所有的中断源在产生中断请求时都需要确定采用哪种中断模式。

如果将 ARM920T 内核中的程序状态寄存器 CPSR 的 F 位置 1,则不能接收来自中断控制器的 FIQ 中断;如果 CPSR 的 I 位被置 1,则不能接收来自中断控制器的 IRQ 中断。所

以，通过将 CPSR 中的 F 位、I 位及中断屏蔽寄存器 INTMSK 中的相应位清 0，中断控制器可以接收中断。

S3C2440A 的中断控制器支持的 60 个中断源如表 4-21 所示。

表 4-21　S3C2440A 的中断控制器支持的 60 个中断源

中断源	中断源描述	仲裁组
INT_ADC	ADC EOC 和 Touch 中断(INT_ADC_S/INT_TC)	ARB5
INT_RTC	RTC 报警中断	
INT_SPI1	SPI1 中断	
INT_UART0	UART0 中断(ERR、RXD 和 TXD)	
INT_IIC	IIC 中断	ARB4
INT_USBH	USB 主设备中断	
INT_USBD	USB 从设备中断	
INT_NFCON	Nand Flash 控制器中断	
INT_UART1	UART1 中断(ERR、RXD、TXD)	
INT_SPI0	SPI0 中断	
INT_SDI	SDI 中断	ARB3
INT_DMA3	DMA 通道 3 中断	
INT_DMA2	DMA 通道 2 中断	
INT_DMA1	DMA 通道 1 中断	
INT_DMA0	DMA 通道 0 中断	
INT_LCD	LCD 中断(INT_FrSyn、INT_FiCnt)	
INT_UART2	UART2 中断(ERR、RXD、TXD)	ARB2
INT_TIMER4	Timer4 中断	
INT_TIMER3	Timer3 中断	
INT_TIMER2	Timer2 中断	
INT_TIMER1	Timer1 中断	
INT_TIMER0	Timer0 中断	
INT_WDT_AC97	Watch-Dog 定时器中断(INT_WDT、INT_AC97)	ARB1
INT_TICK	RTC Time tick 中断	
nBATT_FLT	Battery Fault 中断	
INT_CAM	Camera 接口(INT_CAM_C，INT_CAM_P)	
EINT8_23	External 中断 8～23	
EINT4_7	External 中断 4～7	
EINT3	External 中断 3	ARB0
EINT2	External 中断 2	
EINT1	External 中断 1	
EINT0	External 中断 0	

有些中断源含有若干个不同类型的子中断源对应不同类型的操作，如 INT_UART1 包含 ERR、RXD、TXD 共 3 种操作，发生 INT_UART1 后需要进一步进行区分。子中断源的

信息如表 4-22 所示。

表 4-22　子中断源的信息

子中断源	子中断源描述	中断源
INT_AC97	AC97 中断	INT_WDT_AC97
INT_WDT	看门狗中断	
INT_CAM_P	P-port 摄像头接口捕捉中断	INT_CAM
INT_CAM_C	C-port 摄像头接口捕捉中断	
INT_ADC_S	ADC 中断	INT_ADC
INT_TC	触摸屏中断(笔上/下)	
INT_ERR2	UART2 错误中断	INT_UART2
INT_TXD2	UART2 发送中断	
INT_RXD2	UART2 接收中断	
INT_ERR1	UART1 错误中断	INT_UART1
INT_TXD1	UART1 发送中断	
INT_RXD1	UART1 接收中断	
INT_ERR0	UART0 错误中断	INT_UART0
INT_TXD0	UART0 发送中断	
INT_RXD0	UART0 接收中断	

5. 中断优先级及中断优先级产生模块

若嵌入式系统中有多个中断源,需要对各个中断源进行优先级设置,原因如下。

(1) 多个中断源可能会同时产生中断请求,需要择优。

(2) 当正在处理某一中断时,有可能会出现新的中断请求,需要确定是否响应新中断请求。

中断优先级的设置通常通过硬件实现,S3C2440A 中的中断控制器采用优先级编码电路实现优先级逻辑。在多个中断源同时请求中断时,硬件优先级逻辑可以决定哪一个中断应该得到响应;然后,这个硬件逻辑产生一个跳转指令跳到向量表中对应的中断向量地址处,在这个地址上事先已经放置了跳转到与该中断相应的中断服务程序的跳转指令,极大地减小了中断延迟。

S3C2440A 的 60 个中断源通过 32 个外部中断引脚产生中断请求,因此有些中断源需要共用引脚。这 32 个中断请求在优先级逻辑电路中先输入一级仲裁器进行第一次裁决,裁决的结果继续输入到二级仲裁器实现最终裁决——筛选出最高优先级的中断申请。S3C2440A 采用的优先级编码电路包括基于仲裁器的 7 个翻转:图 4-6 所示的 6 个一级仲裁器和 1 个二级仲裁器。

每个仲裁器基于一位仲裁器模式控制信号(ARB_MODE)和两位选择控制信号(ARB_SEL)来处理 6 个中断请求。

(1) 如果 ARB_SEL 位是 00b,优先级是 REQ0,REQ1,REQ2,REQ3,REQ4,REQ5。

(2) 如果 ARB_SEL 位是 01b,优先级是 REQ0,REQ2,REQ3,REQ4,REQ1,REQ5。

(3) 如果 ARB_SEL 位是 10b,优先级是 REQ0,REQ3,REQ4,REQ1,REQ2,REQ5。

(4) 如果 ARB_SEL 位是 11b,优先级是 REQ0,REQ4,REQ1,REQ2,REQ3,REQ5。

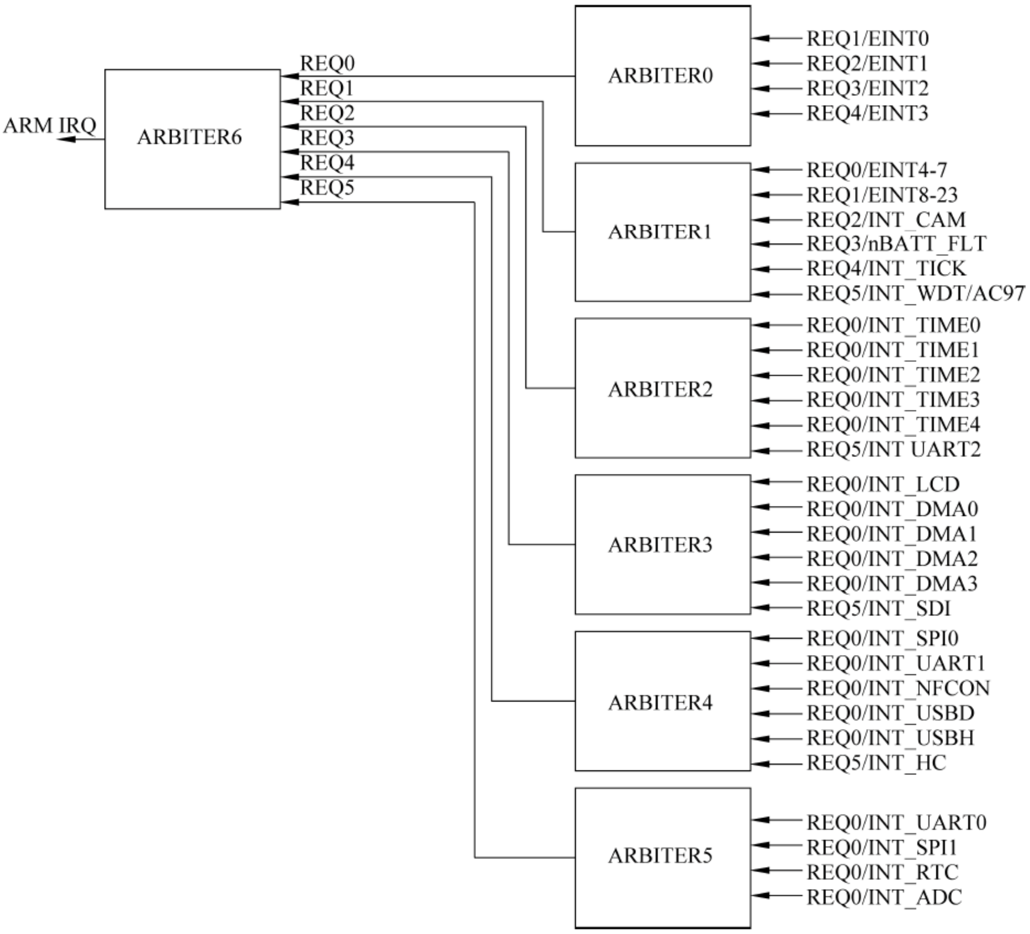


图 4-6 S3C2440A 中断优先级生成模块

> **注意** 仲裁器的 REQ0 总是有最高优先级,REQ5 总是有最低优先级。此外,通过改变 ARB_SEL 位,可以翻转 REQ1～REQ4 的优先级。

如果将 ARB_MODE 位置 0,ARB_SEL 位不会自动改变,使得仲裁器在一个固定优先级的模式下操作(注意在此模式下,通过手工改变 ARB_SEL 位来配置优先级)。另外,如果 ARB_MODE 位是 1,ARB_SEL 位以翻转的方式改变。例如,如果 REQ1 被服务,则 ARB_SEL 位自动变为 01b,把 REQ1 放到最低的优先级。ARB_SEL 变化的详细规则如下。

(1) 如果 REQ0 或 REQ5 被服务,ARB_SEL 位完全不会变化。

(2) 如果 REQ1 被服务,ARB_SEL 位变为 01b。

(3) 如果 REQ2 被服务,ARB_SEL 位变为 10b。

(4) 如果 REQ3 被服务,ARB_SEL 位变为 11b。

(5) 如果 REQ4 被服务,ARB_SEL 位变为 00b。

这 7 个仲裁器的 ARB_MODE(1 位)和 ARB_SEL(2 位)共同构成 21 位中断优先级寄存器。

仲裁过程依赖于硬件优先级逻辑且其结果被写入中断未决寄存器,通报那些由不同中

断源生成的中断。

6. S3C2440A 内中断控制器的寄存器

用 S3C2440A 的中断方式来控制 I/O 端口或部件操作时，除了要对 I/O 端口或部件的相应寄存器进行初始化设置外，还需对中断控制器的控制寄存器进行初始化设置。

在中断控制器中有 5 个控制寄存器：中断源未决寄存器、中断模式寄存器、中断屏蔽寄存器、优先级寄存器和中断未决寄存器。各个寄存器间的关系如图 4-7 所示。

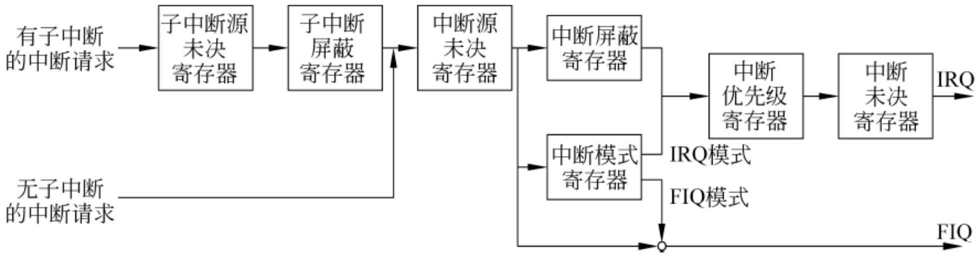


图 4-7 S3C2440A 各个中断控制寄存器间的关系

(1) 中断源未决寄存器

中断源未决寄存器(SRCPND)包括 32 位，每位对应一个中断源。如果相应的中断源产生中断请求且等待中断服务，则对应位置 1。因此这个寄存器可以指出哪个中断源在请求服务。注意，SRCPND 的每个位都由中断源自动置位，不管中断屏蔽 INTMASK 寄存器是否对其屏蔽。此外，SRCPND 寄存器也不会受到中断控制器优先级逻辑的影响。

在中断源对应的中断服务程序中，SRCPND 的相应位必须被清除，以便下次能正确得到同一中断源的中断请求。如果从中断服务程序返回没有清除该位，中断控制器将视其为同一个中断源的又一次中断请求。换言之，如果 SRCPND 的某位被置 1，则其将总被当成一个有效的中断请求在等待服务。清除相应位的时间依赖于用户需求。如果想再次收到来自同一个中断源的有效请求，就应该清除相应的位，然后使能中断。对 SRCPND 寄存器的位进行置 1 或清除的操作比较特殊：可以通过对相应位写 1 来清除相应位；如果对相应位写 0，则该位的数值保持不变。

SRCPND 寄存器的地址是 0x4A000000，可读写，初始值为 0x00000000。各位的定义如表 4-23 所示。

表 4-23 SRCPND 寄存器各位的定义

SRCPND 对应的中断源	位	描　述	初始值
INT_ADC	[31]	0=未请求　1=已请求	0
INT_RTC	[30]	0=未请求　1=已请求	0
INT_SPI1	[29]	0=未请求　1=已请求	0
INT_UART0	[28]	0=未请求　1=已请求	0
INT_IIC	[27]	0=未请求　1=已请求	0
INT_USBH	[26]	0=未请求　1=已请求	0
INT_USBD	[25]	0=未请求　1=已请求	0
INT_NFCON	[24]	0=未请求　1=已请求	0

续表

SRCPND 对应的中断源	位	描　述	初始值
INT_UART1	[23]	0=未请求　1=已请求	0
INT_SPI0	[22]	0=未请求　1=已请求	0
INT_SDI	[21]	0=未请求　1=已请求	0
INT_DMA3	[20]	0=未请求　1=已请求	0
INT_DMA2	[19]	0=未请求　1=已请求	0
INT_DMA1	[18]	0=未请求　1=已请求	0
INT_DMA0	[17]	0=未请求　1=已请求	0
INT_LCD	[16]	0=未请求　1=已请求	0
INT_UART2	[15]	0=未请求　1=已请求	0
INT_TIMER4	[14]	0=未请求　1=已请求	0
INT_TIMER3	[13]	0=未请求　1=已请求	0
INT_TIMER2	[12]	0=未请求　1=已请求	0
INT_TIMER1	[11]	0=未请求　1=已请求	0
INT_TIMER0	[10]	0=未请求　1=已请求	0
INT_WDT_AC97	[9]	0=未请求　1=已请求	0
INT_TICK	[8]	0=未请求　1=已请求	0
nBATT_FLT	[7]	0=未请求　1=已请求	0
INT_CAM	[6]	0=未请求　1=已请求	0
EINT8_23	[5]	0=未请求　1=已请求	0
EINT4_7	[4]	0=未请求　1=已请求	0
EINT3	[3]	0=未请求　1=已请求	0
EINT2	[2]	0=未请求　1=已请求	0
EINT1	[1]	0=未请求　1=已请求	0
EINT0	[0]	0=未请求　1=已请求	0

(2) 中断模式寄存器

S3C2440A 的中断模式有两种：FIQ 模式和 IRQ 模式。所有中断源的中断请求首先都是在中断源未决寄存器中登记的，这些中断请求需要确定中断模式。

中断模式寄存器(INTMOD)包括 32 位，每位与一个中断源相关。如果某位为 1，则相应的中断将在 FIQ 模式下处理；如果某位为 0，则在 IRQ 模式下操作。中断源提出中断请求并确定中断模式后，中断能否得到响应，需要看当前程序状态寄存器 CPSR 的 F 位和 I 位的值：若 F 位为 0，则可以响应 FIQ 中断，若 F 位为 1，则屏蔽 FIQ 中断；若 I 位为 0，则可以响应 IRQ 中断，若 I 位为 1，则屏蔽 IRQ 中断。

注意　仅有一个中断源能在 FIQ 模式下服务，也就是说，INTMOD 中仅有一位可以被置 1。

INTMOD 寄存器的地址是 0x4A000004，可读写，初始值为 0x00000000。各位的定义如表 4-24 所示。

表 4-24 INTMOD 寄存器各位的定义

INTMOD 对应的中断源	位	描 述	初始值
INT_ADC	[31]	0=IRQ 模式 1=FIQ 模式	0
INT_RTC	[30]	0=IRQ 模式 1=FIQ 模式	0
INT_SPI1	[29]	0=IRQ 模式 1=FIQ 模式	0
INT_UART0	[28]	0=IRQ 模式 1=FIQ 模式	0
INT_IIC	[27]	0=IRQ 模式 1=FIQ 模式	0
INT_USBH	[26]	0=IRQ 模式 1=FIQ 模式	0
INT_USBD	[25]	0=IRQ 模式 1=FIQ 模式	0
INT_NFCON	[24]	0=IRQ 模式 1=FIQ 模式	0
INT_UART1	[23]	0=IRQ 模式 1=FIQ 模式	0
INT_SPI0	[22]	0=IRQ 模式 1=FIQ 模式	0
INT_SDI	[21]	0=IRQ 模式 1=FIQ 模式	0
INT_DMA3	[20]	0=IRQ 模式 1=FIQ 模式	0
INT_DMA2	[19]	0=IRQ 模式 1=FIQ 模式	0
INT_DMA1	[18]	0=IRQ 模式 1=FIQ 模式	0
INT_DMA0	[17]	0=IRQ 模式 1=FIQ 模式	0
INT_LCD	[16]	0=IRQ 模式 1=FIQ 模式	0
INT_UART2	[15]	0=IRQ 模式 1=FIQ 模式	0
INT_TIMER4	[14]	0=IRQ 模式 1=FIQ 模式	0
INT_TIMER3	[13]	0=IRQ 模式 1=FIQ 模式	0
INT_TIMER2	[12]	0=IRQ 模式 1=FIQ 模式	0
INT_TIMER1	[11]	0=IRQ 模式 1=FIQ 模式	0
INT_TIMER0	[10]	0=IRQ 模式 1=FIQ 模式	0
INT_WDT_AC97	[9]	0=IRQ 模式 1=FIQ 模式	0
INT_TICK	[8]	0=IRQ 模式 1=FIQ 模式	0
nBATT_FLT	[7]	0=IRQ 模式 1=FIQ 模式	0
INT_CAM	[6]	0=IRQ 模式 1=FIQ 模式	0
EINT8_23	[5]	0=IRQ 模式 1=FIQ 模式	0
EINT4_7	[4]	0=IRQ 模式 1=FIQ 模式	0
EINT3	[3]	0=IRQ 模式 1=FIQ 模式	0
EINT2	[2]	0=IRQ 模式 1=FIQ 模式	0
EINT1	[1]	0=IRQ 模式 1=FIQ 模式	0
EINT0	[0]	0=IRQ 模式 1=FIQ 模式	0

(3) 中断屏蔽寄存器

对于采用 FIQ 模式的中断请求,若 CPSR 的 F 位为 0,则中断请求可以按 FIQ 模式进行处理。采用 IRQ 模式的中断请求能否得到响应除了要看 CPSR 的 I 位是否为 0 外,还需要通过中断屏蔽寄存器的“过滤”:确定为 IRQ 模式后,如果中断屏蔽寄存器对应的位为 1,则该中断请求将被屏蔽。

中断屏蔽寄存器(INTMSK)包括 32 位,每位都和一个中断源相对应。如果某位为 1,

则 CPU 不会服务相应中断源的中断请求(注意 SRCPND 的相应位还会被置 1)。如果屏蔽位为 0,中断请求可以被服务。中断屏蔽寄存器只针对 IRQ 模式的中断请求,对 FIQ 模式不起作用。

INTMSK 寄存器的地址是 0x4A000008,可读写,初始值为 0xFFFFFFFF。各位的定义如表 4-25 所示。

表 4-25 INTMSK 寄存器各位的定义

INTMSK 对应的中断源	位	描　述	初始值
INT_ADC	[31]	0=未屏蔽　1=屏蔽	1
INT_RTC	[30]	0=未屏蔽　1=屏蔽	1
INT_SPI1	[29]	0=未屏蔽　1=屏蔽	1
INT_UART0	[28]	0=未屏蔽　1=屏蔽	1
INT_IIC	[27]	0=未屏蔽　1=屏蔽	1
INT_USBH	[26]	0=未屏蔽　1=屏蔽	1
INT_USBD	[25]	0=未屏蔽　1=屏蔽	1
INT_NFCON	[24]	0=未屏蔽　1=屏蔽	1
INT_UART1	[23]	0=未屏蔽　1=屏蔽	1
INT_SPI0	[22]	0=未屏蔽　1=屏蔽	1
INT_SDI	[21]	0=未屏蔽　1=屏蔽	1
INT_DMA3	[20]	0=未屏蔽　1=屏蔽	1
INT_DMA2	[19]	0=未屏蔽　1=屏蔽	1
INT_DMA1	[18]	0=未屏蔽　1=屏蔽	1
INT_DMA0	[17]	0=未屏蔽　1=屏蔽	1
INT_LCD	[16]	0=未屏蔽　1=屏蔽	1
INT_UART2	[15]	0=未屏蔽　1=屏蔽	1
INT_TIMER4	[14]	0=未屏蔽　1=屏蔽	1
INT_TIMER3	[13]	0=未屏蔽　1=屏蔽	1
INT_TIMER2	[12]	0=未屏蔽　1=屏蔽	1
INT_TIMER1	[11]	0=未屏蔽　1=屏蔽	1
INT_TIMER0	[10]	0=未屏蔽　1=屏蔽	1
INT_WDT_AC97	[9]	0=未屏蔽　1=屏蔽	1
INT_TICK	[8]	0=未屏蔽　1=屏蔽	1
nBATT_FLT	[7]	0=未屏蔽　1=屏蔽	1
INT_CAM	[6]	0=未屏蔽　1=屏蔽	1
EINT8_23	[5]	0=未屏蔽　1=屏蔽	1
EINT4_7	[4]	0=未屏蔽　1=屏蔽	1
EINT3	[3]	0=未屏蔽　1=屏蔽	1
EINT2	[2]	0=未屏蔽　1=屏蔽	1
EINT1	[1]	0=未屏蔽　1=屏蔽	1
EINT0	[0]	0=未屏蔽　1=屏蔽	1

(4) 中断优先级寄存器

中断优先级寄存器(PRIORITY)是针对 IRQ 模式下的各中断请求进行优先级设置的寄存器，每个中断源有 3 位对应，分别代表仲裁器一位仲裁器模式控制信号(ARB_MODE)和两位选择控制信号(ARB_SEL)。

PRIORITY 寄存器的地址是 0x4A00000C，可读写，初始值为 0x7F。各位的定义如表 4-26 所示。

表 4-26 PRIORITY 寄存器各位的定义

PRIORITY	位	描 述	初始值
ARB_SEL6	[20:19]	仲裁器组 6 优先级顺序集 00=REQ 1—2—3—4，01=REQ 2—3—4—1 10=REQ 3—4—1—2，11=REQ 4—1—2—3	00
ARB_SEL5	[18:17]	仲裁器组 5 优先级顺序集 00=REQ 1—2—3—4，01=REQ 2—3—4—1 10=REQ 3—4—1—2，11=REQ 4—1—2—3	00
ARB_SEL4	[16:15]	仲裁器组 4 优先级顺序集 00=REQ 1—2—3—4，01=REQ 2—3—4—1 10=REQ 3—4—1—2，11=REQ 4—1—2—3	00
ARB_SEL3	[14:13]	仲裁器组 3 优先级顺序集 00=REQ 1—2—3—4，01=REQ 2—3—4—1 10=REQ 3—4—1—2，11=REQ 4—1—2—3	00
ARB_SEL2	[12:11]	仲裁器组 2 优先级顺序集 00=REQ 1—2—3—4，01=REQ 2—3—4—1 10=REQ 3—4—1—2，11=REQ 4—1—2—3	00
ARB_SEL1	[10:9]	仲裁器组 1 优先级顺序集 00=REQ 1—2—3—4，01=REQ 2—3—4—1 10=REQ 3—4—1—2，11=REQ 4—1—2—3	00
ARB_SEL0	[8:7]	仲裁器组 0 优先级顺序集 00=REQ 1—2—3—4，01=REQ 2—3—4—1 10=REQ 3—4—1—2，11=REQ 4—1—2—3	00
ARB_MODE6	[6]	仲裁器组 6 优先级翻转使能 0=优先级不翻转 1=优先级翻转使能	1
ARB_MODE5	[5]	仲裁器组 5 优先级翻转使能 0=优先级不翻转 1=优先级翻转使能	1
ARB_MODE4	[4]	仲裁器组 4 优先级翻转使能 0=优先级不翻转 1=优先级翻转使能	1
ARB_MODE3	[3]	仲裁器组 3 优先级翻转使能 0=优先级不翻转 1=优先级翻转使能	1
ARB_MODE2	[2]	仲裁器组 2 优先级翻转使能 0=优先级不翻转 1=优先级翻转使能	1
ARB_MODE1	[1]	仲裁器组 1 优先级翻转使能 0=优先级不翻转 1=优先级翻转使能	1
ARB_MODE0	[0]	仲裁器组 0 优先级翻转使能 0=优先级不翻转 1=优先级翻转使能	1

(5) 中断未决寄存器

中断未决寄存器(INTPND)是 32 位寄存器，每一位对应一个中断源。只有未被屏蔽且

有最高优先级、在源未决寄存器中进行申请的中断请求可以将其对应的中断未决位置 1，表示可以得到响应。该寄存器只对 IRQ 模式有效。因为 INTPND 寄存器位于优先级逻辑之后，所以仅有 1 位可以被置 1，同时中断控制器产生 IRQ 信号给 ARM920T 核。此时，只要 ARM920T 核内部的当前程序状态寄存器的 I 标志被清零，对应的中断服务程序就可以开始执行。在 IRQ 模式的中断服务程序中，可以读取中断未决寄存器确定哪个中断源正在被服务。

在中断服务程序中中断未决寄存器的中断未决位（即 1）必须清零，以避免由于同一个中断未决位导致中断服务程序反复执行。同中断源未决寄存器类似，可以通过对相应位写 1 来进行清零；对相应位写 0 进行保持。因此，清除中断未决寄存器的未决位最简单快捷的方式就是将中断未决寄存器的值写回到中断未决寄存器里。

INTPND 寄存器的地址是 0x4A000010，可读写，初始值为 0x00000000。各位的定义如表 4-27 所示。

表 4-27　INTPND 寄存器各位的定义

INTPND 对应的中断源	位	描　述	初始值
INT_ADC	[31]	0=未请求　1=已请求	0
INT_RTC	[30]	0=未请求　1=已请求	0
INT_SPI1	[29]	0=未请求　1=已请求	0
INT_UART0	[28]	0=未请求　1=已请求	0
INT_IIC	[27]	0=未请求　1=已请求	0
INT_USBH	[26]	0=未请求　1=已请求	0
INT_USBD	[25]	0=未请求　1=已请求	0
INT_NFCON	[24]	0=未请求　1=已请求	0
INT_UART1	[23]	0=未请求　1=已请求	0
INT_SPI0	[22]	0=未请求　1=已请求	0
INT_SDI	[21]	0=未请求　1=已请求	0
INT_DMA3	[20]	0=未请求　1=已请求	0
INT_DMA2	[19]	0=未请求　1=已请求	0
INT_DMA1	[18]	0=未请求　1=已请求	0
INT_DMA0	[17]	0=未请求　1=已请求	0
INT_LCD	[16]	0=未请求　1=已请求	0
INT_UART2	[15]	0=未请求　1=已请求	0
INT_TIMER4	[14]	0=未请求　1=已请求	0
INT_TIMER3	[13]	0=未请求　1=已请求	0
INT_TIMER2	[12]	0=未请求　1=已请求	0
INT_TIMER1	[11]	0=未请求　1=已请求	0
INT_TIMER0	[10]	0=未请求　1=已请求	0
INT_WDT_AC97	[9]	0=未请求　1=已请求	0
INT_TICK	[8]	0=未请求　1=已请求	0
nBATT_FLT	[7]	0=未请求　1=已请求	0

续表

INTPND 对应的中断源	位	描　述	初始值
INT_CAM	[6]	0＝未请求　1＝已请求	0
EINT8_23	[5]	0＝未请求　1＝已请求	0
EINT4_7	[4]	0＝未请求　1＝已请求	0
EINT3	[3]	0＝未请求　1＝已请求	0
EINT2	[2]	0＝未请求　1＝已请求	0
EINT1	[1]	0＝未请求　1＝已请求	0
EINT0	[0]	0＝未请求　1＝已请求	0

以上 5 个寄存器是 S3C2440A 中断控制器中主要的寄存器，在处理每个中断源时，设计者应根据实际需要通过编程进行设定，即确定将寄存器中的每一位设置为 0 还是设置为 1。除了这几个寄存器外还有如下几个寄存器，在中断源进行控制时，有时需要使用。

(6) 中断偏移寄存器

中断偏移寄存器(INTOFFSET)中的值代表了中断源号，即在 IRQ 模式下，INTPND 寄存器中的某位置 1，则 INTOFFSET 寄存器中的值是其对应中断源的偏移量。该寄存器可以通过清除 SRCPND 寄存器和 INTPND 寄存器的未决位被自动清除。

该寄存器只对 IRQ 模式下的中断有效。

INTOFFSET 寄存器的地址是 0x4A000014，可读，初始值为 0x00000000。各位的定义如表 4-28 所示。

表 4-28　INTOFFSET 寄存器各位的定义

INTOFFSET 对应的中断源	偏移量	INTOFFSET 对应的中断源	偏移量
INT_ADC	31	INT_UART2	15
INT_RTC	30	INT_TIMER4	14
INT_SPI1	29	INT_TIMER3	13
INT_UART0	28	INT_TIMER2	12
INT_IIC	27	INT_TIMER1	11
INT_USBH	26	INT_TIMER0	10
INT_USBD	25	INT_WDT_AC97	9
INT_NFCON	24	INT_TICK	8
INT_UART1	23	nBATT_FLT	7
INT_SPI0	22	INT_CAM	6
INT_SDI	21	EINT8_23	5
INT_DMA3	20	EINT4_7	4
INT_DMA2	19	EINT3	3
INT_DMA1	18	EINT2	2
INT_DMA0	17	EINT1	1
INT_LCD	16	EINT0	0

(7) 子中断源未决寄存器

子中断源未决寄存器(SUBSRCPND)用于那些共用中断请求信号的中断源控制,其作用和操作与 SRCPND 寄存器相同。

该寄存器的地址是 0x4A000018,可读写,初始值为 0x00000000。各位的定义如表 4-29 所示。

表 4-29　SUBSRCPND 寄存器各位的定义

SUBSRCPND 对应的中断源	位	描　述	初始值
保留	[31:15]	—	0
INT_AC97	[14]	0=未请求　1=已请求	0
INT_WDT	[13]	0=未请求　1=已请求	0
INT_CAM_P	[12]	0=未请求　1=已请求	0
INT_CAM_C	[11]	0=未请求　1=已请求	0
INT_ADC_S	[10]	0=未请求　1=已请求	0
INT_TC	[9]	0=未请求　1=已请求	0
INT_ERR2	[8]	0=未请求　1=已请求	0
INT_TXD2	[7]	0=未请求　1=已请求	0
INT_RXD2	[6]	0=未请求　1=已请求	0
INT_ERR1	[5]	0=未请求　1=已请求	0
INT_TXD1	[4]	0=未请求　1=已请求	0
INT_RXD1	[3]	0=未请求　1=已请求	0
INT_ERR0	[2]	0=未请求　1=已请求	0
INT_TXD0	[1]	0=未请求　1=已请求	0
EINT0	[0]	0=未请求　1=已请求	0

(8) 子中断屏蔽寄存器

子中断屏蔽寄存器(INTSUBMSK)也用于那些共用中断请求信号的中断源控制,该寄存器有 11 位,其作用和操作与 SRCPND 寄存器相同,每位和一个中断源相关。如果某位为 1,则 CPU 不会服务相应中断源的中断请求。如果屏蔽位为 0,中断请求可以被服务。

子中断屏蔽寄存器的地址是 0x4A00001C,可读,初始值为 0x000007FF。各位的定义如表 4-30 所示。

表 4-30　INTSUBMSK 寄存器各位的定义

SUBINTMSK 对应的中断源	位	描　述	初始值
保留	[31:15]	—	0
INT_AC97	[14]	0=服务有用　1=服务屏蔽	1
INT_WDT	[13]	0=服务有用　1=服务屏蔽	1
INT_CAM_P	[12]	0=服务有用　1=服务屏蔽	1
INT_CAM_C	[11]	0=服务有用　1=服务屏蔽	1
INT_ADC_S	[10]	0=服务有用　1=服务屏蔽	1
INT_TC	[9]	0=服务有用　1=服务屏蔽	1

续表

SUBINTMSK 对应的中断源	位	描　述	初始值
INT_ERR2	[8]	0=服务有用　1=服务屏蔽	1
INT_TXD2	[7]	0=服务有用　1=服务屏蔽	1
INT_RXD2	[6]	0=服务有用　1=服务屏蔽	1
INT_ERR1	[5]	0=服务有用　1=服务屏蔽	1
INT_TXD1	[4]	0=服务有用　1=服务屏蔽	1
INT_RXD1	[3]	0=服务有用　1=服务屏蔽	1
INT_ERR0	[2]	0=服务有用　1=服务屏蔽	1
INT_TXD0	[1]	0=服务有用　1=服务屏蔽	1
INT_RXD0	[0]	0=服务有用　1=服务屏蔽	1

7. 中断编程

中断编程涉及中断向量表的建立、CPSR 中的 F 位和 I 位设置、用户堆栈设置、中断向量设置、中断控制寄存器初始化及中断服务程序编写等部分。其中，中断向量表的建立和 CPSR 设置及用户堆栈设置通常在系统启动过程中完成，对于现成的嵌入式系统硬件平台来讲是不需要用户编写的。用户主要进行中断向量设置、初始化中断控制寄存器及编写中断服务程序。

```
;*************************************** (*******************************************
;Description:    汇编语言定义宏 HANDLER,实现现场保护和转向中断服务程序
;************************************* (*********************************************
    MACRO
$HandlerLabel HANDLER $HandleLabel
$HandlerLabel
    sub   sp,sp,#4              ;sp 值减小,准备将跳转地址入栈
    stmfd  sp!,{r0}             ;工作寄存器内容入栈(lr 为原程序返回地址,不入栈)
    LDR   r0,=$HandleLabel      ;加载 HandleXXX 到 r0 中
    LDR   r0,[r0]               ;加载 HandleXXX 的内容(中断服务程序地址,即中断向量)
    str   r0,[sp,#4]            ;HandleXXX 对应的中断向量入栈
    ldmfd  sp!,{r0,pc}          ;工作寄存器内容出栈,中断向量弹入 PC(跳转到 ISR)
    MEND
```

在系统引导程序中定义了中断向量表，例如：

```
;*********************************************************************************
    ;Description:部分系统引导程序
;*********************************************************************************
    ;在 0 地址处定义中断向量表
    b     ResetHandler
    b     HandlerUndef           ;未定义指令异常
    b     HandlerSWI             ;软中断异常
    b     HandlerPabort          ;指令预取中止异常
    b     HandlerDabort          ;数据中止异常
    b     .                      ;保留
    b     HandlerIRQ             ;IRQ 中断异常
    b     HandlerFIQ             ;FIQ 中断异常
```

```
…
;各个异常跳转地址处对应的宏调用
HandlerFIQ       HANDLER    HandleFIQ
HandlerIRQ       HANDLER    HandleIRQ
HandlerUndef     HANDLER    HandleUndef
HandlerSWI       HANDLER    HandleSWI
HandlerDabort    HANDLER    HandleDabort
HandlerPabort    HANDLER    HandlePabort
…
;初始化中断向量对应的内存空间
IsrIRQ
    sub   sp,sp,#4
    stmfd  sp!,{r8-r9}
    ldr   r9,=INTOFFSET            ;INTOFFSET=0x4A000014
    ldr   r9,[r9]
    ldr   r8,=HandleEINT0          ;取得存储 EINT0 中断向量的存储地址作为基地址
    add   r8,r8,r9,lsl #2;
    ldr   r8,[r8]
    str   r8,[sp,#8]
    ldmfd  sp!,{r8-r9,pc}
;存储各种中断向量的内存空间定义
^  _ISR_STARTADDRESS               ;_ISR_STARTADDRESS=0x33FFFF00
    HandleReset      #   4         ;为复位中断向量预留 4B
    HandleUndef      #   4         ;为未定义中断向量预留 4B,以下类似
    HandleSWI        #   4
    HandlePabort     #   4
    HandleDabort     #   4
    HandleReserved   #   4
    HandleIRQ        #   4
    HandleFIQ        #   4
    HandleEINT0      #   4         ;为 EINT0 中断向量预留 4B
    HandleEINT1      #   4         ;为 EINT1 中断向量预留 4B,以下类似
    HandleEINT2      #   4
    HandleEINT3      #   4
    HandleEINT4_7    #   4
    HandleEINT8_23   #   4         ;EINT8~23 共用同一中断向量,预留 4B
    …
```

当系统启动时,在没有操作系统的系统中,系统初始化由系统引导程序完成,所以在进入应用程序前就完成了中断向量所在存储空间的初始化。在中断发生前只要正确地将中断向量存储到对应的存储空间中(位于从 0x33FFFF00 开始的存储区内)即可实现从主程序转向中断服务程序。如采用中断方式进行串行发送的部分程序如下:

```
#define rINTPND      (*(volatile unsigned*)0x4a000010)
#define rSUBSRCPND   (*(volatile unsigned*)0x4a000018)
#define rINTSUBMSK   (*(volatile unsigned*)0x4a00001c)
#define rSRCPND      (*(volatile unsigned*)0x4a000000)
#define rUCON0       (*(volatile unsigned*)0x50000004)
#define rULCON0      (*(volatile unsigned*)0x50000000)
#define rUTRSTAT0    (*(volatile unsigned*)0x50000010)
```

```
#define rGPHCON      (*(volatile unsigned*)0x56000070)
                                                 //H端口控制寄存器,见 4.3.4 小节
#define rGPHUP      (*(volatile unsigned*)0x56000078)     //H端口上拉电阻寄存器
#define rGPGCON      (*(volatile unsigned*)0x56000060)     //G端口控制寄存器
#define rGPGUP      (*(volatile unsigned*)0x56000068)     //G端口上拉电阻寄存器
#define pISR_UART0 (*(unsigned*)(_ISR_STARTADDRESS+0x90))   //0x33FFFF00+0x90
#define WrUTXH0(ch) (*(volatile unsigned char*)0x50000020)=(unsigned char)(ch)
    static int whichUart=0;                              //通道号
    void Test_Uart0_Int(void)
    {
        volatile static char*uart0TxStr;
        unsigned char ch;
        unsigned int isDone,isTxInt,isRxInt;
        int iBaud;
        rGPHCON&=0x3c0000;
        rGPHCON|=0x2aaaa;                              //所有的 UART 通道使能
        rGPHUP|=0x1ff;                                 //UART 端口 H上拉电阻禁止
        rGPGCON|=(0xf<<18);                            //nRTS1,nCTS1
        rGPGUP|=(0x3<<9);                              //UART 端口 G上拉电阻禁止
        Uart_Printf("\nConnect PC[COM1 or COM2] and UART0 of SMDK2440 \n");
        Uart_Printf("Then,press any key...\n");
        /*******************UART0 Tx test with interrupt*******************/
        isTxInt=1;
        uart0TxStr="ABCDEF1234567890->UART0 Tx interrupt test is good!!!!\r\n";
        Uart_Printf("[Uart channel 0 Tx Interrupt Test]\n");
        pISR_UART0=(unsigned)Uart0_TxInt;     //中断向量填充到相应的存储空间
        rULCON0=(0<<6)|(0<<3)|(0<<2)|(3);     //正常,无校验,1位停止,8位数据
        rUCON0|=(1<<9)|(1<<8)|(0<<7)|(0<<6)|(0<<5)|(0<<4)|(1<<2)|(1);
        if(ch==0)
            while(!(rUTRSTAT0 & 0x4));        //等待 com0 发送移位寄存器空
        else if(ch==1)
            while(!(rUTRSTAT1 & 0x4));        //等待 com1 发送移位寄存器空
        else if(ch==2)
            while(!(rUTRSTAT2 & 0x4));        //等待 com2 发送移位寄存器空
        rINTMSK=~(0x1<<28);
        rINTSUBMSK=~(0x1<<1);
        while(isTxInt);                       //等待中断发生
        …//其他功能语句
}
```

Uart_Printf 为串口输出函数,限于篇幅这里不再展开。发送中断服务函数代码如下:

```
void __irq Uart0_TxInt(void)
{
    rINTSUBMSK|=(1|2|4);
      if(*uart0TxStr !='\0')
      {
          WrUTXH0(*uart0TxStr++);
          ClearPending((0x1<<28));          //清主中断源未决位
          rSUBSRCPND=(0x1<<1);              //清子中断源未决位
          rINTSUBMSK&=~(2);                 //不屏蔽子中断
```

```
        }
        else
        {
            isTxInt=0;
            ClearPending((0x1<<28));
            rSUBSRCPND=((0x1<<1));
            rINTMSK|=((0x1<<28));
        }
}
__inline void ClearPending(int bit)
{
    register i;
    rSRCPND=bit;
    rINTPND=bit;
    i=rINTPND;
}
__inline void ClearSubPending(int bit)
{
    register i;
    rSUBSRCPND=bit;
    i=rINTPND;
}
```

4.3.4 通用 I/O 口

任务：掌握 S3C2440A 的通用 I/O 口的功能、结构及初始化。

在嵌入式系统的输入输出设计中，如果是少量数据的输入输出(如控制其他装置的按键信号)，可以使用通用 I/O 口——通用输入/输出端口，即 GPIO 引脚。GPIO 可以通过程序设定来控制，但不适合传输大量数据。

S3C2440A 提供了 130 个 GPIO 引脚，分为 9 组端口，GPA～GPH、GPJ，用户可将每个端口通过软件配置为输入模式、输出模式或特殊功能模式，以满足不同的系统配置和设计要求。

控制 S3C2440A 的 GPIO 端口的寄存器有 3 类：GPxCON、GPxDAT、GPxUP(x＝A～H、J，但没有 GPAUP)。

GPxCON：GPIO 控制寄存器，可以设置选定 GPIO 口的输入输出方式和功能。GPA 组的 23 个端口比较特殊，只能是输出方式。GPACON 的每一位对应一个引脚，当某位为 0 时，对应引脚为输出端口，否则为复用功能。

GPB～GPH、GPJ 端口的 GPxCON 寄存器使用方法一致，每两位控制一个引脚，00 表示输入 I/O 口；01 表示输出 I/O 口；10 表示复用功能；11 表示保留。

GPxDAT：用于读写引脚的状态，即端口数据。当引脚配置为输出时，给该寄存器的某位写 1，则对应引脚输出高电平；写 0 则输出低电平。当引脚配置为输出时，读该寄存器可得到端口电平状态。

GPxUP：设置引脚是否使用上拉电阻；某位为 0 时对应引脚使用上拉电阻；某位为 1 时不使用上拉电阻。注意：GPA 组没有 GPxUP 寄存器，即没有上拉电阻。

必须对整体资源进行规划，先确定引脚在主程序中的功能，然后对对应的端口进行功能配置，避免应用时出现问题。

1. 端口寄存器

(1) 端口 A 的寄存器

端口 A 有两个寄存器：控制寄存器 GPACON 和数据寄存器 GPADAT。

GPACON 可读写，端口地址为 0x56000000，初始值为 0xFFFFFF，各位的定义如表 4-31 所示。

表 4-31 GPACON 各位的定义

GPACON	位	功能描述	GPACON	位	功能描述
GPA24	[24]	保留	GPA11	[11]	0=输出，1=ADDR26
GPA23	[23]	保留	GPA10	[10]	0=输出，1=ADDR25
GPA22	[22]	0=输出，1=nFCE	GPA9	[9]	0=输出，1=ADDR24
GPA21	[21]	0=输出，1=nRSTOUT	GPA8	[8]	0=输出，1=ADDR23
GPA20	[20]	0=输出，1=nFRE	GPA7	[7]	0=输出，1=ADDR22
GPA19	[19]	0=输出，1=nFWE	GPA6	[6]	0=输出，1=ADDR21
GPA18	[18]	0=输出，1=ALE	GPA5	[5]	0=输出，1=ADDR20
GPA17	[17]	0=输出，1=CLE	GPA4	[4]	0=输出，1=ADDR19
GPA16	[16]	0=输出，1=nGCS5	GPA3	[3]	0=输出，1=ADDR18
GPA15	[15]	0=输出，1=nGCS4	GPA2	[2]	0=输出，1=ADDR17
GPA14	[14]	0=输出，1=nGCS3	GPA1	[1]	0=输出，1=ADDR16
GPA13	[13]	0=输出，1=nGCS2	GPA0	[0]	0=输出，1=ADDR0
GPA12	[12]	0=输出，1=nGCS1			

GPADAT 可读写，端口地址为 0x56000004，初始值未定义，各位的定义如表 4-32 所示。

表 4-32 GPADAT 各位的定义

GPADAT	位	功能描述
GPA[24:0]	[24:0]	如果端口配置为输出端口，写到 GPADAT 寄存器的数据可以输出到各个位对应的引脚上；如果端口配置为输入端口，则能从对应引脚读入相应的外部输入数据

(2) 端口 B 的寄存器

端口 B 有 3 个寄存器：控制寄存器 GPBCON、数据寄存器 GPBDAT 和上拉电阻寄存器 GPBUP。

GPBCON 可读写，端口地址为 0x56000010，初始值为 0x0，各位的定义如表 4-33 所示。

表 4-33 GPBCON 各位的定义

GPBCON	位	功能描述
GPB10	[21:20]	00=输入，01=输出，10=nXDREQ0，11=Reserved
GPB9	[19:18]	00=输入，01=输出，10=nXDACK0，11=Reserved
GPB8	[17:16]	00=输入，01=输出，10=nXDREQ1，11=Reserved
GPB7	[15:14]	00=输入，01=输出，10=nXDACK1，11=Reserved
GPB6	[13:12]	00=输入，01=输出，10=nXBREQ，11=Reserved
GPB5	[11:10]	00=输入，01=输出，10=nXBACK，11=Reserved
GPB4	[9:8]	00=输入，01=输出，10=TCLK0，11=Reserved
GPB3	[7:6]	00=输入，01=输出，10=TOUT3，11=Reserved

续表

GPBCON	位	功 能 描 述
GPB2	[5:4]	00＝输入,01＝输出,10＝TOUT2,11＝Reserved
GPB1	[3:2]	00＝输入,01＝输出,10＝TOUT1,11＝Reserved
GPB0	[1:0]	00＝输入,01＝输出,10＝TOUT0,11＝Reserved

GPBDAT 可读写,端口地址为 0x56000014,初始值未定义,各位的定义如表 4-34 所示。

表 4-34　GPBDAT 各位的定义

GPBDAT	位	功 能 描 述
GPB[10:0]	[10:0]	如果端口配置为输出端口,写到 GPBDAT 寄存器的数据可以输出到各个位对应的引脚上;如果端口配置为输入端口,能从对应引脚读入相应的外部输入数据;如果该引脚被设置为功能引脚,则读到的数据不确定

GPBUP 可读写,端口地址为 0x56000018,初始值为 0x0,各位的定义如表 4-35 所示。

表 4-35　GPBUP 各位的定义

GPBUP	位	功 能 描 述
GPB[10:0]	[10:0]	0＝允许端口 B 的相应引脚上接上拉电阻 1＝禁止端口 B 的相应引脚上接上拉电阻

(3) 端口 C 的寄存器

端口 C 有 3 个寄存器:控制寄存器 GPCCON、数据寄存器 GPCDAT 和上拉电阻寄存器 GPCUP。

GPCCON 可读写,端口地址为 0x56000020,初始值为 0x0,各位的定义如表 4-36 所示。

表 4-36　GPCCON 各位的定义

GPCCON	位	功 能 描 述
GPC15	[31:30]	00＝输入,01＝输出,10＝VD7,11＝Reserved
GPC14	[29:28]	00＝输入,01＝输出,10＝VD6,11＝Reserved
GPC13	[27:26]	00＝输入,01＝输出,10＝VD5,11＝Reserved
GPC12	[25:24]	00＝输入,01＝输出,10＝VD4,11＝Reserved
GPC11	[23:22]	00＝输入,01＝输出,10＝VD3,11＝Reserved
GPC10	[21:20]	00＝输入,01＝输出,10＝VD2,11＝Reserved
GPC9	[19:18]	00＝输入,01＝输出,10＝VD1,11＝Reserved
GPC8	[17:16]	00＝输入,01＝输出,10＝VD0,11＝Reserved
GPC7	[15:14]	00＝输入,01＝输出,10＝LCD_LPCREVB,11＝Reserved
GPC6	[13:12]	00＝输入,01＝输出,10＝LCD_LPCREV,11＝Reserved
GPC5	[11:10]	00＝输入,01＝输出,10＝LCD_LPCOE,11＝Reserved
GPC4	[9:8]	00＝输入,01＝输出,10＝VM,11＝I2SSDI
GPC3	[7:6]	00＝输入,01＝输出,10＝VFRAME,11＝Reserved
GPC2	[5:4]	00＝输入,01＝输出,10＝VLINE,11＝Reserved
GPC1	[3:2]	00＝输入,01＝输出,10＝VCLK,11＝Reserved
GPC0	[1:0]	00＝输入,01＝输出,10＝LEND,11＝Reserved

GPCDAT 可读写,端口地址为 0x56000024,初始值未定义,各位的定义如表 4-37 所示。

表 4-37 GPCDAT 各位的定义

GPCDAT	位	功能描述
GPC[15:0]	[15:0]	如果端口配置为输出端口,写到 GPCDAT 寄存器的数据可以输出到各个位对应的引脚上;如果端口配置为输入端口,能从对应引脚读入相应的外部输入数据;如果该引脚被设置为功能引脚,则读到的数据不确定

GPCUP 可读写,端口地址为 0x56000028,初始值为 0x0,各位的定义如表 4-38 所示。

表 4-38 GPCUP 各位的定义

GPCUP	位	功能描述
GPC[15:0]	[15:0]	0=允许端口 C 的相应引脚上接上拉电阻 1=禁止端口 C 的相应引脚上接上拉电阻

(4) 端口 D 的寄存器

端口 D 有 3 个寄存器:控制寄存器 GPDCON、数据寄存器 GPDDAT 和上拉电阻寄存器 GPDUP。

GPDCON 可读写,端口地址为 0x56000030,初始值为 0x0,各位的定义如表 4-39 所示。

表 4-39 GPDCON 各位的定义

GPDCON	位	功能描述
GPD15	[31:30]	00=输入,01=输出,10=VD23,11=nSS0
GPD14	[29:28]	00=输入,01=输出,10=VD22,11=nSS1
GPD13	[27:26]	00=输入,01=输出,10=VD21,11=Reserved
GPD12	[25:24]	00=输入,01=输出,10=VD20,11=Reserved
GPD11	[23:22]	00=输入,01=输出,10=VD19,11=Reserved
GPD10	[21:20]	00=输入,01=输出,10=VD18,11=SPICLK1
GPD9	[19:18]	00=输入,01=输出,10=VD17,11=SPIMOSI1
GPD8	[17:16]	00=输入,01=输出,10=VD16,11=SPIMISO1
GPD7	[15:14]	00=输入,01=输出,10=VD15,11=Reserved
GPD6	[13:12]	00=输入,01=输出,10=VD14,11=Reserved
GPD5	[11:10]	00=输入,01=输出,10=VD13,11=Reserved
GPD4	[9:8]	00=输入,01=输出,10=VD12,11=Reserved
GPD3	[7:6]	00=输入,01=输出,10=VD11,11=Reserved
GPD2	[5:4]	00=输入,01=输出,10=VD10,11=Reserved
GPD1	[3:2]	00=输入,01=输出,10=VD9,11=Reserved
GPD0	[1:0]	00=输入,01=输出,10=VD8,11=Reserved

GPDDAT 可读写,端口地址为 0x56000034,初始值未定义,各位的定义如表 4-40 所示。

表 4-40 GPDDAT 各位的定义

GPDDAT	位	功能描述
GPD[15:0]	[15:0]	如果端口配置为输出端口,写到 GPDDAT 寄存器的数据可以输出到各个位对应的引脚上;如果端口配置为输入端口,能从对应引脚读入相应的外部输入数据;如果该引脚被设置为功能引脚,则读到的数据不确定

GPDUP 可读写,端口地址为 0x56000038,初始值为 0xF000,各位的定义如表 4-41 所示。

表 4-41 GPDUP 各位的定义

GPDUP	位	功 能 描 述
GPD[15:0]	[15:0]	0=允许端口 D 的相应引脚上接上拉电阻 1=禁止端口 D 的相应引脚上接上拉电阻

(5) 端口 E 的寄存器

端口 E 有 3 个寄存器:控制寄存器 GPECON、数据寄存器 GPEDAT 和上拉电阻寄存器 GPEUP。

GPECON 可读写,端口地址为 0x56000040,初始值为 0x0,各位的定义如表 4-42 所示。

表 4-42 GPECON 各位的定义

GPECON	位	功 能 描 述
GPE15	[31:30]	00=输入,01=输出,10=IICSDA,11=Reserved
GPE14	[29:28]	00=输入,01=输出,10=IICSCL,11=Reserved
GPE13	[27:26]	00=输入,01=输出,10=SPICLK0,11=Reserved
GPE12	[25:24]	00=输入,01=输出,10=SPIMOSI0,11=Reserved
GPE11	[23:22]	00=输入,01=输出,10=SPIMISO0,11=Reserved
GPE10	[21:20]	00=输入,01=输出,10=SDDAT3,11=Reserved
GPE9	[19:18]	00=输入,01=输出,10=SDDAT2,11=Reserved
GPE8	[17:16]	00=输入,01=输出,10=SDDAT1,11=Reserved
GPE7	[15:14]	00=输入,01=输出,10=SDDAT0,11=Reserved
GPE6	[13:12]	00=输入,01=输出,10=SDCMD,11=Reserved
GPE5	[11:10]	00=输入,01=输出,10=SDCLK,11=Reserved
GPE4	[9:8]	00=输入,01=输出,10=I2SDO,11=AC_SDATA_OUT
GPE3	[7:6]	00=输入,01=输出,10=I2SDI,11=AC_SDATA_IN
GPE2	[5:4]	00=输入,01=输出,10=CDCLK,11=AC_nRESET
GPE1	[3:2]	00=输入,01=输出,10=I2SSCLK,11=AC_BIT_CLK
GPE0	[1:0]	00=输入,01=输出,10=I2SLRCK,11=AC_SYNC

GPEDAT 可读写,端口地址为 0x56000044,初始值未定义,各位的定义如表 4-43 所示。

表 4-43 GPEDAT 各位的定义

GPEDAT	位	功 能 描 述
GPE[15:0]	[15:0]	如果端口配置为输出端口,写到 GPEDAT 寄存器的数据可以输出到各个位对应的引脚上;如果端口配置为输入端口,能从对应引脚读入相应的外部输入数据;如果该引脚被设置为功能引脚,则读到的数据不确定

GPEUP 可读写,端口地址为 0x56000048,初始值为 0x0,各位的定义如表 4-44 所示。

表 4-44　GPEUP 各位的定义

GPEUP	位	功能描述
GPE[13:0]	[13:0]	0=允许端口 E 的相应引脚上接上拉电阻 1=禁止端口 E 的相应引脚上接上拉电阻

(6) 端口 F 的寄存器

端口 F 有 3 个寄存器：控制寄存器 GPFCON、数据寄存器 GPFDAT 和上拉电阻寄存器 GPFUP。

如果 GPF0～GPF7 用于省电模式下的唤醒信号，端口需要设置成中断模式。

GPFCON 可读写，端口地址为 0x56000050，初始值为 0x0，各位的定义如表 4-45 所示。

表 4-45　GPFCON 各位的定义

GPFCON	位	功能描述
GPF7	[15:14]	00=输入，01=输出，10=EINT7，11=Reserved
GPF6	[13:12]	00=输入，01=输出，10=EINT6，11=Reserved
GPF5	[11:10]	00=输入，01=输出，10=EINT5，11=Reserved
GPF4	[9:8]	00=输入，01=输出，10=EINT4，11=Reserved
GPF3	[7:6]	00=输入，01=输出，10=EINT3，11=Reserved
GPF2	[5:4]	00=输入，01=输出，10=EINT2，11=Reserved
GPF1	[3:2]	00=输入，01=输出，10=EINT1，11=Reserved
GPF0	[1:0]	00=输入，01=输出，10=EINT0，11=Reserved

GPFDAT 可读写，端口地址为 0x56000054，初始值未定义，各位的定义如表 4-46 所示。

表 4-46　GPFDAT 各位的定义

GPFDAT	位	功能描述
GPF[7:0]	[7:0]	如果端口配置为输出端口，写到 GPFDAT 寄存器的数据可以输出到各个位对应的引脚上；如果端口配置为输入端口，能从对应引脚读入相应的外部输入数据；如果该引脚被设置为功能引脚，则读到的数据不确定

GPFUP 的端口地址为 0x56000058，初始值为 0x0，各位的定义如表 4-47 所示。

表 4-47　GPFUP 各位的定义

GPFUP	位	功能描述
GPF[7:0]	[7:0]	0=允许端口 F 的相应引脚上接上拉电阻 1=禁止端口 F 的相应引脚上接上拉电阻

(7) 端口 G 的寄存器

端口 G 有 3 个寄存器：控制寄存器 GPGCON、数据寄存器 GPGDAT 和上拉电阻寄存器 GPGUP。如果 GPG0～GPG7 用于睡眠模式下的唤醒信号，端口需要设置成中断模式。

GPGCON 可读写，端口地址为 0x56000060，初始值为 0x0，各位的定义如表 4-48 所示。

注意　GPG[15:13]在 Nand Flash 启动模式下必须作为输入。

表 4-48 GPGCON 各位的定义

GPGCON	位	功能描述
GPG15	[31:30]	00=输入,01=输出,10=EINT23,11=Reserved
GPG14	[29:28]	00=输入,01=输出,10=EINT22,11=Reserved
GPG13	[27:26]	00=输入,01=输出,10=EINT21,11=Reserved
GPG12	[25:24]	00=输入,01=输出,10=EINT20,11=Reserved
GPG11	[23:22]	00=输入,01=输出,10=EINT19,11=TCLK1
GPG10	[21:20]	00=输入,01=输出,10=EINT18,11=nCTS1
GPG9	[19:18]	00=输入,01=输出,10=EINT17,11=nRTS1
GPG8	[17:16]	00=输入,01=输出,10=EINT16,11=Reserved
GPG7	[15:14]	00=输入,01=输出,10=EINT15,11=SPICLK1
GPG6	[13:12]	00=输入,01=输出,10=EINT14,11=SPIMOSI1
GPG5	[11:10]	00=输入,01=输出,10=EINT13,11=SPIMISO1
GPG4	[9:8]	00=输入,01=输出,10=EINT12,11=LCD_PWRDN
GPG3	[7:6]	00=输入,01=输出,10=EINT11,11=nSS1
GPG2	[5:4]	00=输入,01=输出,10=EINT10,11=nSS0
GPG1	[3:2]	00=输入,01=输出,10=EINT9,11=Reserved
GPG0	[1:0]	00=输入,01=输出,10=EINT8,11=Reserved

GPGDAT 可读写,端口地址为 0x56000064,初始值未定义,各位的定义如表 4-49 所示。

表 4-49 GPGDAT 各位的定义

GPGDAT	位	功能描述
GPG[15:0]	[15:0]	如果端口配置为输出端口,写到 GPGDAT 寄存器的数据可以输出到各个位对应的引脚上;如果端口配置为输入端口,能从对应引脚读入相应的外部输入数据;如果该引脚被设置为功能引脚,则读到的数据不确定

GPGUP 可读写,端口地址为 0x56000068,初始值为 0xFC00,各位的定义如表 4-50 所示。

表 4-50 GPGUP 各位的定义

GPGUP	位	功能描述
GPG[15:0]	[15:0]	0=允许端口 G 的相应引脚上接上拉电阻 1=禁止端口 G 的相应引脚上接上拉电阻

(8) 端口 H 的寄存器

端口 H 有 3 个寄存器:控制寄存器 GPHCON、数据寄存器 GPHDAT 和上拉电阻寄存器 GPHUP。

GPHCON 可读写,端口地址为 0x56000070,初始值为 0x0,各位的定义如表 4-51 所示。

表 4-51 GPHCON 各位的定义

GPHCON	位	功 能 描 述
GPH10	[21:20]	00=输入,01=输出,10=CLKOUT1,11=Reserved
GPH9	[19:18]	00=输入,01=输出,10=CLKOUT0,11=Reserved
GPH8	[17:16]	00=输入,01=输出,10=UEXTCLK,11=Reserved
GPH7	[15:14]	00=输入,01=输出,10=RXD2,11=nCTS1
GPH6	[13:12]	00=输入,01=输出,10=TXD2,11=nRTS1
GPH5	[11:10]	00=输入,01=输出,10=RXD1,11=Reserved
GPH4	[9:8]	00=输入,01=输出,10=TXD1,11=Reserved
GPH3	[7:6]	00=输入,01=输出,10=RXD0,11=Reserved
GPH2	[5:4]	00=输入,01=输出,10=TXD0,11=Reserved
GPH1	[3:2]	00=输入,01=输出,10=nRTS0,11=Reserved
GPH0	[1:0]	00=输入,01=输出,10=nCTS0,11=Reserved

GPHDAT 可读写,端口地址为 0x56000074,初始值未定义,各位的定义如表 4-52 所示。

表 4-52 GPHDAT 各位的定义

GPHDAT	位	功 能 描 述
GPH[15:0]	[15:0]	如果端口配置为输出端口,写到 GPHDAT 寄存器的数据可以输出到各个位对应的引脚上;如果端口配置为输入端口,能从对应引脚读入相应的外部输入数据;如果该引脚被设置为功能引脚,则读到的数据不确定

GPHUP 可读写,端口地址为 0x56000078,初始值为 0x0,各位的定义如表表 4-53 所示。

表 4-53 GPHUP 各位的定义

GPHUP	位	功 能 描 述
GPH[10:0]	[10:0]	0=允许端口 H 的相应引脚上接上拉电阻 1=禁止端口 H 的相应引脚上接上拉电阻

(9) 端口 J 的寄存器

端口 J 有 3 个寄存器:控制寄存器 GPJCON、数据寄存器 GPJDAT 和上拉电阻寄存器 GPJUP。

GPJCON 可读写,端口地址为 0x560000D0,初始值为 0x0,各位的定义如表 4-54 所示。

表 4-54 GPJCON 各位的定义

GPJCON	位	功 能 描 述
GPJ12	[23:22]	00=输入,01=输出,10=CAMRESET,11=Reserved
GPJ11	[21:20]	00=输入,01=输出,10=CAMCLKOUT,11=Reserved
GPJ10	[19:18]	00=输入,01=输出,10=CAMHREF,11=Reserved
GPJ9	[17:16]	00=输入,01=输出,10=CAMVSYNC,11=Reserved
GPJ8	[15:14]	00=输入,01=输出,10=CAMPCLK,11=Reserved
GPJ7	[13:12]	00=输入,01=输出,10=CAMDATA7,11=Reserved

续表

GPJCON	位	功能描述
GPJ6	[11:10]	00=输入,01=输出,10=CAMDATA6,11=Reserved
GPJ5	[9:8]	00=输入,01=输出,10=CAMDATA5,11=Reserved
GPJ4	[7:6]	00=输入,01=输出,10=CAMDATA4,11=Reserved
GPJ3	[5:4]	00=输入,01=输出,10=CAMDATA3,11=Reserved
GPJ2	[3:2]	00=输入,01=输出,10=CAMDATA2,11=Reserved
GPJ1	[1:0]	00=输入,01=输出,10=CAMDATA1,11=Reserved
GPJ0	[21:20]	00=输入,01=输出,10=CAMDATA0,11=Reserved

GPJDAT 可读写,端口地址为 0x560000D4,初始值未定义,各位的定义如表 4-55 所示。

表 4-55 GPJDAT 各位的定义

GPJDAT	位	功能描述
GPJ[12:0]	[12:0]	如果端口配置为输出端口,写到 GPJDAT 寄存器的数据可以输出到各个位对应的引脚上;如果端口配置为输入端口,能从对应引脚读入相应的外部输入数据;如果该引脚被设置为功能引脚,则读到的数据不确定

端口 J 上拉电阻寄存器 GPJUP 可读写,端口地址为 0x560000D8,初始值 0x0000,各位的定义如表 4-56 所示。

表 4-56 GPJUP 各位的定义

GPJUP	位	功能描述
GPJ[12:0]	[12:0]	0=允许端口 J 的相应引脚上接上拉电阻 1=禁止端口 J 的相应引脚上接上拉电阻

2. GPIO 设置实例

在系统中端口 F 的 4～7 引脚上依次连接了 LED,使用下列程序段完成相关寄存器的赋值,控制 LED 是否点亮。若输出信号为低电平则 LED 点亮;反之,LED 不亮。在下列程序段中,先同时点亮 GPF[7:6]上连接的 LED,再同时点亮 GPF[5:4]上连接的 LED。

```
;******************************************************************
;Description:    点亮端口 F 连接的 LED
;******************************************************************
Led_Test
    mov     r0,#0x56000000        ;取端口 A 数据寄存器的地址作为基地址
    mov     r1,#0x5500
    str     r1,[r0,#0x50]         ;相对寻址,将 0x5500 存入 GPFCON,设置 GPF[7:4]为输出
    mov     r1,#0x30
    str     r1,[r0,#0x54]         ;相对寻址,将 0x30 存入 GPFDAT,点亮 GPF[7:6]的 LED
    mov     r2,#0x100000          ;延时
0   subs    r2,r2,#1
    bne     %B0
    mov     r1,#0xc0
    str     r1,[r0,#0x54]         ;相对寻址,将 0xc0 存入 GPFDAT,点亮 GPF[5:4]的 LED
    mov     pc,lr
```

4.3.5 串行通信

任务：掌握 S3C2440A 的 UART 的功能、操作、结构及初始化。

在嵌入式系统的输入输出设计中，GPIO 可以通过程序设定来控制，但不适合传输大量数据。传输大量数据时，通常采用异步串行通信的方式。

异步串行通信技术简单，但非常重要。它提供了一种用于数据通信的最简单的标准接口，因此在很多设备上都得到了广泛的应用。在对复杂的嵌入式系统进行调试的时候，经常会利用串口作为控制台，使得普通 PC 可以通过串口来控制系统，实现人机交互。例如，前面已经使用过的利用串口终端工具观察程序运行情况的方法。

异步串行通信是以帧的方式传输数据的。每一帧有效数据前有 1 个起始位(0)，帧结束有 1 个或多个停止位(1)。也就是说，为保证同步，异步串行通信传输的数据帧中含有起始位和停止位作为同步信息。通常，每帧中含有 5、6、7 或 8 位数据。传输线在空闲状态时保持逻辑 1。开始传输时，先发送一个起始位，使得传输线上的信号从逻辑 1 跳变到 0，表明数据帧传输开始。接收端则在检测到起始位后，按照收发两端事先约定好的通信速度，检测后面的数据位，从而组成一帧数据。一帧数据是从最低有效位开始传输的。在传输的最后，利用 1 个或多个停止位使传输线回到空闲状态，然后发送方才可以继续发送下一帧数据。可见，数据帧中的起始位和停止位保证了数据传输的同步。

图 4-8 为含 8 位有效数据的异步串行传输数据帧格式，无奇偶校验(N)，含 1 位停止位，即常用的 8N1。

数据位时钟	1	2	3	4	5	6	7	8	9	10
发送端信号	起始位	8位数据								停止位

图 4-8 典型的异步串行传输数据帧格式

在异步通信过程中，接收方需要知道每一位的传输时间——速率，该传输速率称为波特率，单位为波特 baud，1baud=1 位/秒(bps)。只要发送方和接收方采用一致的波特率，就能可靠地完成异步传输。

1. RS-232C 标准

RS-232C 标准是应用广泛的异步串行通信接口，是美国 EIA 与 BELL 等公司一起开发的通信协议，适合于数据传输速率在 0～20000bps 范围内的通信。RS-232C 标准最初是为远程通信连接数据终端设备 DTE(Data Terminal Equipment)而定制的，当时并未考虑计算机系统的应用要求，但目前又被广泛应用于计算机与终端或外围设备间的近端连接标准。这个标准的有些规定和计算机系统是不一致甚至是矛盾的。

RS-232C 标准包括了硬件协议，它用于连接两种设备：DTE 和 DCE（Data Communication Equipment，数据通信设备）。RS-232C 标准定义了接口的机械特性、电气特性和功能特性。

最初，RS-232C 标准可采用 DB-25 和 DB-9 连接器，在实际应用中大量使用的是 DB-9。DB-9 的外形和接口引脚如图 4-9 所示。

RS-232C 标准接口有 25 条线，DB-9 只设置了常用的 9 根，具体说明如下。

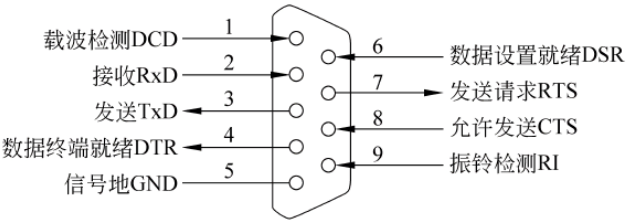


图 4-9　PC 端 RS-232C 接口引脚定义

(1) 状态线

DSR：数据准备就绪，有效时表明数据通信设备可用。

DTR：数据终端就绪，有效时表明数据终端设备可用。

这两个信号有时连到电源上，上电后立即有效，但这两个状态信号有效只表示设备本身可用，并不说明通信链路可以开始进行通信了，能否开始进行通信要由下面的控制信号决定。

(2) 联络线

RTS：请求发送，DTE 准备向 DCE 发送数据时，使该信号有效，通知 DCE 要发送数据给 DCE 了。

CTS：允许发送，是对 RTS 的响应信号。当 DCE 已准备好接收 DTE 传来的数据时，使该信号有效，通知 DTE 开始发送数据。

RTS/CTS 请求应答联络信号用于实现半双工 Modem 系统中发送方式和接收方式之间的切换。在全双工系统中，因为配置了双向通道，故不需要 RTS/CTS 联络信号，使其变高。

(3) 数据线

TxD：发送数据。DTE 通过该引脚发送数据到 DCE。

RxD：接收数据。DCE 发送数据到 DTE，即 DTE 通过该引脚接收 DCE 发送的数据。

(4) 其余

DCD：载波检测，用来表示 DCE 已接通通信链路，告知 DTE 准备接收数据。

RI：振铃检测，当 DCE 收到交换台发送来的振铃呼叫信号时，使该信号有效，通知 DTE 已被呼叫。

GND：信号地。

在通常的应用系统中，往往是在 CPU 和 I/O 设备之间传送信息，两者都是 DTE，例如，PC 和色温计、PC 和单片机之间的通信，双方都能发送和接收，只需使用 3 根线即可，即双方的 RxD 和 TxD 与对方的 TxD 和 RxD 交叉连接，以及接地线 GND。

通过 RS-232C 标准连接 DTE 和 DCE 时，因为这两种设备的接口是配对连接的，可以直接连接，且它们的引脚定义也是配对的：TxD 和 RxD 连接；RTS 和 CTS 连接；DSR 和 DTR 连接。

在电气方面，RS-232C 标准规定：

在 TxD 和 RxD 上——−15～−3V 为逻辑 1；+3～+15V 为逻辑 0。

在 RTS、CTS、DSR、DTR 和 DCD 等控制线上——信号有效(接通，ON 状态，正电压)用+3～+15V 表示；信号无效(断开，OFF 状态，负电压)用−15～−3V 表示。

由以上定义可以看出，信号无效的电平低于−3V，也就是当传输电平的绝对值大于3V 时，电路可以有效地检查出来，介于−3～+3V 之间的电压无意义，低于−15V 或高于

+15V 的电压也被认为无意义,因此,实际工作时,应保证电平的绝对值在+3~+15V 之间。

当计算机和 TTL 电平的设备通信时,如计算机和单片机通信时,需要使用 RS-232C/TTL 电平转换器件,常用的有 MAX232。

2. S3C2440A 的 UART 的基本功能

S3C2440A 内部集成的 UART(Universal Asynchronous Receiver or Transmitter,通用异步收发器)单元提供 3 个独立的异步串行 I/O 通道,也就是通常所说的串口,这些接口用于支持异步串行通信。每个 I/O 通道都可以在中断或 DMA 两种模式下工作。在系统时钟下,支持的最高波特率为 230.4Kbps。每个 UART 通道包含 2 个 64 位 FIFO 分别供接收和发送使用。

S3C2440A 的 UART 具有以下主要功能。

(1) 设定传输波特率

异步串行通信要求双方的数据传输速率相同。波特率可通过由时钟源(PCLK、FCLK 或 UEXTCLK)16 分频和 UART 波特率除数寄存器(UBRDIVn)指定的 16 位除数决定。只要收发双方采用一致的波特率,就能可靠地完成异步通信。

(2) 数据的串行、并行转换

将接收到的串行数据变换为并行数据;将需要发送的并行数据转换为输出的串行数据。

(3) 设定数据帧格式

异步串行通信的数据帧可以包括 1 位起始位(0)、5~8 位数据位、0 或 1 位奇偶校验位、1~2 位停止位(1)。S3C2440A 的 UART 可以根据传输需要进行数据位数、停止位数和奇偶校验位的设置。

(4) 奇偶校验

若在数据帧中设置了采用奇偶校验方式,则 S3C2440A 的 UART 可以实现发送端数据的奇偶校验位设置,接收方 UART 则可对接收到的数据进行相同规则的验证,判断数据是否正确。

(5) 收发数据缓冲

S3C2440A 的 UART 内部有用于收发缓冲的缓冲区——64 字节的 FIFO。

每个 UART 模块包含以下几个部件:波特率发生器、发送器、接收器和控制单元。

波特率发生器可以以 PCLK、FCLK 或 UEXTCLK(外部输入时钟)作为时钟源。发送器和接收器包含 64 字节的 FIFO 和移位器。待发送的数据,首先写入 FIFO,然后复制到发送移位器中,最后从数据输出端口(TxDn)依次被移位输出。被接收的数据也同样从数据接收端口(RxDn)依次被移位输入到接收移位器,当接收到一个字节时就复制到 FIFO 中。

S3C2440A 的 UART 内部结构如图 4-10 所示。

3. S3C2440A 的 UART 操作

S3C2440A 的 UART 操作包括数据发送、数据接收、中断发生、波特率发生、回送模式以及自动流控制等内容。

(1) 数据发送

数据发送的帧格式是可编程的。它包含一个起始位、5~8 个数据位、1 个可选的奇偶校验位和 1~2 个停止位,这些都可以通过线控制寄存器(ULCONn)设置。发送器也能够产生发送中止条件。中止条件强制串口输出逻辑 0 达一帧的时间,它在一个字传输完成以后阻塞停止信号的传送。中止信号发送之后,继续传输数据到发送 FIFO 中(在非 FIFO 模式

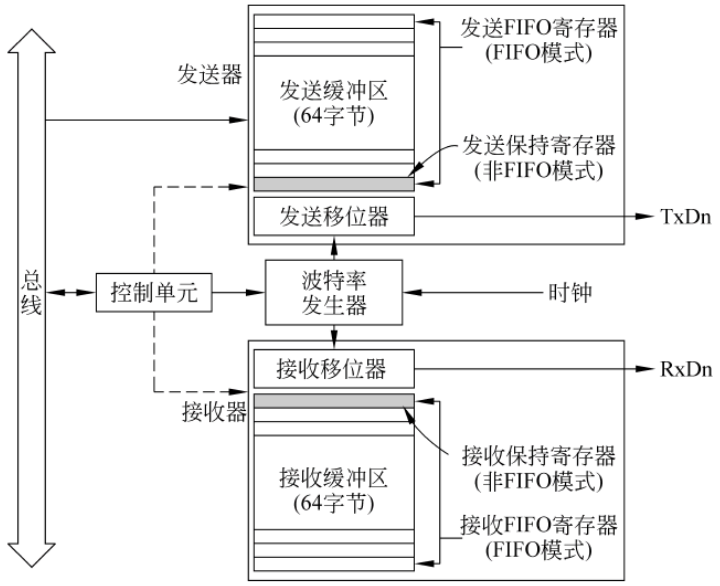


图 4-10　S3C2440A 的 UART 内部结构

下，将被放到发送保持寄存器中）。

（2）数据接收

与发送一样，接收的数据帧格式同样是可编程的。它包括了一个起始位、5～8 个数据位、1 个可选的奇偶校验位和 1～2 个停止位，这些也可以通过线控制寄存器（ULCONn）来设置。接收器可以检测到溢出错误、奇偶校验错误、帧错误和中止状况，在每种情况下都会将一个错误标志置位。检测到的各种错误的描述如下。

① 溢出错误表示新的数据已经覆盖了还没有来得及取出的旧数据，也就是说接收数据速度快于取出的速度导致一部分数据还没有取出就被新的数据覆盖了。

② 奇偶校验错误表示接收器检测到了意料之外的奇偶校验结果。

③ 帧错误表示接收到的数据没有有效的停止位。

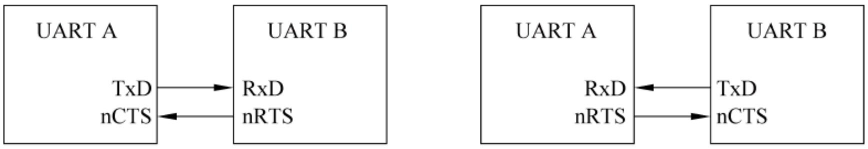
④ 中止状况表示 RxDn 的输入被保持为 0 状态的时间超过了一个帧传输的时间。

⑤ 在 FIFO 模式下，接收 FIFO 不应为空，但当接收器在 3 个字时间内都没有接收到任何数据时，就认为发生了接收超时状况。

（3）自动流控制（AFC）

S3C2440A 的 UART0 和 UART1 通过 nRTS 和 nCTS 信号支持自动流控制，在这种情况下必须将 UART 与 UART 连接。如果用户将 UART 连接到调制解调器，就应该在 UMCONn 寄存器中禁用自动流控制位，并通过软件控制 nRTS。在 AFC 中，nRTS 由接收器的接收情况来控制，nCTS 则控制了发送器的工作。UART 发送器在 nCTS 信号被置 1 时发送 FIFO 中的数据（在 AFC 中，nCTS 意味着对方 UART 的 FIFO 已经准备好接收数据）。在 UART 接收数据时，如果它的接收 FIFO 中还有多于 2 个字节的空余空间就必须将 nRTS 置 1，以便告诉对方“接收准备好”；当接收 FIFO 的剩余空间少于 1 字节时，必须将 nRTS 清 0，说明“不能再接收”。UART AFC 接口如图 4-11 所示。

UART2 不支持 AFC 功能，因为 S3C2440A 没有 nRTS2 和 nCTS2。



(a) UART A发送数据　　(b) UART A接收数据

图 4-11　UART AFC 接口

如果用户要连接到调制解调器接口，就需要使用 nRTS、nCTS、nDSR、nDTR、DCD 和 nRI 信号。在这种情况下，用户可以通过使用其他 I/O 口来由软件控制这些信号，因为 AFC 是不支持 RS-232C 接口的。

(4) 非自动流控制(通过软件控制 nRTS 和 nCTS)

接收操作的步骤如下。

① 选择接收模式(中断或 DMA 模式)。

② 检查 UFSTATn 寄存器中接收 FIFO 计数器的值。如果值小于 32，用户必须设置 UMCONn 第 0 位的值为 1(即激活 nRTS)；如果它等于或大于 32，用户必须设置该位的值为 0(即禁用 nRTS)。

③ 重复第 2 步。

发送操作的步骤如下。

① 选择发送模式(中断或 DMA 模式)。

② 检查 UMSTATn 第 0 位的值，如果为 1(nCTS 被激活)，用户就可以写数据到发送缓冲区或发送 FIFO 寄存器中。

(5) 中断/DMA 请求产生器

S3C2440A 的每个 UART 都有 7 个状态信号：接收 FIFO/缓冲区数据准备好、发送 FIFO/缓冲区空、发送移位器空、溢出错误、奇偶校验错误、帧错误和中止，所有这些状态都由对应的 UART 状态寄存器(UTRSTATn/UERSTATn)中的相应位来表示。

当接收器要将接收移位器的数据传送到接收 FIFO 时，它会激活接收 FIFO 满状态信号，如果控制寄存器中的接收模式被设置为中断模式，就会引发接收中断。

当发送器从发送 FIFO 中取出数据传送到发送移位寄存器时，FIFO 空状态信号将会被激活。如果控制寄存器中的发送模式被设置为中断模式，就会引发发送中断。

如果接收/发送模式被设置为 DMA 模式，“接收 FIFO 满”和“发送 FIFO 空”状态信号同样可以产生 DMA 请求信号。

溢出错误、奇偶校验错误、帧错误和中止状况都被认为是接收错误状态，如果 UART 控制寄存器 UCONn 中的“接收错误状态中断使能位”被置位，则每一种错误都能够引发接收错误中断。当“接收错误状态中断请求”被检测到时，产生请求的信号可以通过读取 UERSTATn 来识别。

(6) UART 错误状态 FIFO

除了接收 FIFO 寄存器之外，UART 还具有一个错误状态 FIFO。错误状态 FIFO 标识在 FIFO 寄存器中接收到的数据是否有错误。

在数据传输过程中，若在接收字符时发生错误，错误中断不会立刻产生，而是直到含有

错误的字符被 CPU 读取时才产生。

为了清除错误状态 FIFO,有错误的接收缓冲寄存器 URXHn 和错误状态寄存器 UERSTATn 必须被读出。

(7) 波特率发生器

每个 UART 的波特率发生器为传输提供了串行移位时钟。波特率发生器的时钟源可以通过 S3C2440A 的串口时钟源来选择。波特率时钟由通过时钟源的 16 分频及一个由 UART 波特率除数寄存器(UBRDIVn)指定的 16 位除数决定。UBRDIVn 的值可以按照下式确定:

UBRDIVn=(取整)(MCLK/(波特率×16))−1

除数的范围为 1～($2^{16}-1$)。例如,如果波特率为 115200bps,且系统主频(MCLK)为 64MHz,则 UBRDIVn 的值为

UBRDIVn=(取整)(64000000/(115200×16))−1=35−1=34

(8) 回送模式

S3C2440A 的 UART 提供了一个测试模式,即回送模式。在这种模式下,发送出的数据会被立即接收。这一特性用于校验运行处理器内部发送和接收通道的功能。该模式可以通过设置 UART 控制寄存器(UCONn)中的回送位来实现。

(9) 红外通信模式

S3C2440A 的 UART 模块支持红外线(IR)发送和接收。可以通过设置 UART 线控制寄存器(ULCONn)中的红外模式位来选择这一模式。

在 IR 发送模式下,发送时通过正常串行发送占空比 3/16 的脉冲波进行调制(当传送的数据位为 0 值时);在 IR 接收时模式下,接收时必须通过检测 3/16 脉冲波来识别 0 值,即在 IR 模式下的 0 值脉宽只是正常模式下的 3/16。

4. UART 接口寄存器

寄存器名称中的 n 表示 0、1 或 2,对应 UART 的通道号。例如,ULCONn 对应通道 0 为 ULCON0;对应通道 1 为 ULCON1。

(1) UART 线控制寄存器 ULCONn

前面几次提到了线控制寄存器,它的作用是规定数据帧格式和选择红外模式。

ULCONn 共有 3 个,端口地址分别为 0x50000000、0x50004000、0x50008000,可读写,初始值为 0x00,各位的定义如表 4-57 所示。

表 4-57　UART 线控制寄存器各位的定义

ULCONn	位	功能描述
Reserved	[7]	保留
Infra-Red Mode	[6]	决定是否使用红外模式 0=正常模式操作　1=红外接收发送模式
Parity Mode	[5:3]	奇偶码校验的生成和检验类型 0xx=无校验　100=奇校验　101=偶校验 110=校验强制/检查为 1　111=校验强制/检查为 0
Stop Bit	[2]	定义停止位位数 0=1 位停止位　1=2 位停止位
Word Length	[1:0]	每帧的数据位数 00=5 位　01=6 位　10=7 位　11=8 位

(2) UART 控制寄存器 UCONn

UCONn 寄存器可以设置 UART 工作在哪一种方式下。UCONn 共有 3 个，每个 UART 接口通道对应一个，端口地址分别为 0x50000004、0x50004004、0x50008004，均可读写，初始值为 0x00，各位的定义如表 4-58 所示。

表 4-58 UART 控制寄存器各位的定义

UCONn	位	功 能 描 述
FCLK divider	[15:12]	当 UART 时钟源选为 FCLK/n 时的分频器值 n 由 UCON0[15:12]、UCON1[15:12]、UCON2[14:12]来决定 UCON2[15]是 FCLK/n 时钟使能位：0=使 FCLK/n 时钟无效，1=使能 FCLK/n 时钟 设置 n 为 7～21，使用 UCON0[15:12] 设置 n 为 22～36，使用 UCON1[15:12] 设置 n 为 37～43，使用 UCON2[14:12] 对于 UCON0，UART 时钟=FCLK/(divider+6)，即 divider=1：UART 时钟=FCLK/7，divider=2：UART 时钟=FCLK/8，…，divider=15：UART 时钟=FCLK/21。其 divider>0，UCON1、UCON2 必须为 0 对于 UCON1，UART 时钟=FCLK/(divider+21)，即 divider=1：UART 时钟=FCLK/22，divider=2：UART 时钟=FCLK/23，…，divider=15：UART时钟=FCLK/36。其 divider>0，UCON0、UCON2 必须为 0 对于 UCON2，UART 时钟=FCLK/(divider+36)，即 divider=1：UART 时钟=FCLK/37，divider=2：UART 时钟=FCLK/38，…，divider=7：UART时钟=FCLK/43。其 divider>0，UCON0、UCON1 必须为 0 如果 UCON0/1[15:12]和 UCON2[14:12]都是 0，分频器为 44，则 UART 时钟=FCLK/44，总的除数范围是 7～44
Clock Selection	[11:10]	选择 PCLK、UEXTCLK 或 FCLK/n 00,10=PCLK　01=UEXTCLK　11=FCLK/n UBRDIVn=(int)(所选时钟/(波特率×16))−1 (如果选择 FCLK/n，应该在选择或取消选择 FCLK/n 时加上注释代码)
Tx Interrupt Type	[9]	确定发送中断请求信号的类型 0=脉冲触发　1=电平触发
Rx Interrupt Type	[8]	确定接收中断请求信号的类型 0=脉冲触发　1=电平触发
Rx Time Out Enable	[7]	在 UART FIFO 有效时，使能接收超时中断。该中断是一个接收中断 0=无效　1=有效
Rx Error Status Interrupt Enable	[6]	使能 UART 对异常产生中断，例如，在接收期间中止信号、帧错误、奇偶校验错误和溢出错误 0=不产生接收错误状态中断　1=产生接收错误状态中断
Loopback Mode	[5]	对该位置位将引起 UART 进入回送模式。该模式仅用于测试目的 0=正常操作　1=回送模式
Send Break Signal	[4]	对该位置位将引起 UART 在一个帧时间内发送一个中止信号。在发送中止信号后该位自动清零 0=正常发送　1=发送中止信号
Transmit Mode	[3:2]	确定从发送缓冲区读出数据的模式 (UART Tx Enable/Disable) 00=无效　01=中断请求或查询模式 10=DMA0(仅对 UART0)或 DMA3(仅对 UART2) 11=DMA1(仅对 UART1)

续表

UCONn	位	功能描述
Receive Mode	[1:0]	确定从接收缓冲区读出数据的模式(UART Rx Enable/Disable) 00=无效　　01=中断请求或查询模式 10=DMA0(只对 UART0)或 DMA3(只对 UART2) 11=DMA1 (只对 UART1)

(3) UART FIFO 控制寄存器(UFCONn)

UFCONn 寄存器用于设置是否使用 FIFO、设置各 FIFO 的触发阈值，即发送 FIFO 中有多少个数据时产生中断，接收 FIFO 中有多少个数据时产生中断。在非 FIFO 模式下，FIFO 的深度为 1，可以通过 UTRSTAT 寄存器中的相应位来判断是否发生中断或是否产生 DMA 请求。在 FIFO 模式下，应该检查寄存器 UFSTAT 中的 TxFIFO Count 位和 TxFIFO Full 位，如果达到在 UFCON 中设置的 FIFO 的触发水平，就会发生中断。

UFCONn 共有 3 个，每个 UART 接口通道对应一个，端口地址分别为 0x50000008、0x50004008、0x50008008，可读写，初始值为 0x00，各位的定义如表 4-59 所示。

表 4-59　UART FIFO 控制寄存器各位的定义

UFCONn	位	功能描述
Tx FIFO Trigger Level	[7:6]	确定发送 FIFO 的触发水平 00=空　01=16B　10=32B　11=48B
Rx FIFO Trigger Level	[5:4]	确定接收 FIFO 的触发水平 00=1B　01=8B　10=16B　11=32B
Reserved	[3]	保留
Tx FIFO Reset	[2]	确定在复位后 FIFO 是否自动清除 0=正常　　1=复位后清除发送 FIFO
Rx FIFO Reset	[1]	确定在复位后 FIFO 是否自动清除 0=正常　　1=复位后清除接收 FIFO
FIFO Enable	[0]	确定是否采用 FIFO 0=禁止　　1=使能

(4) UART MODEM 控制寄存器(UMCONn)

UMCONn 共有两个，对应 UART 接口通道 0 和通道 1，端口地址分别为 0x5000000C、0x5000400C，可读写，初始值为 0x00，各位的定义如表 4-60 所示。

表 4-60　UART MODEM 控制寄存器各位的定义

UMCONn	位	功能描述
Reserved	[7:5]	这些位必须为 0
Auto Flow Control(AFC)	[4]	0=无效　　1=有效
Reserved	[3:1]	这些位必须为 0
Request to Send	[0]	如果 AFC 位有效，则该值被忽略。在这种情况下 S3C2440A 将自动控制 nRTS；如果 AFC 位无效，nRTS 必须由软件控制 0=高电平(不激活 nRTS)　1=低电平(激活 nRTS)

(5) UART 接收发送状态寄存器(UTRSTATn)

UTRSTATn 共有 3 个，每个 UART 接口通道对应一个，端口地址分别为 0x50000010、

0x50004010、0x50008010，只读，初始值为 0x6，各位的定义如表 4-61 所示。

表 4-61 UART 接收发送状态寄存器各位的定义

UTRSTATn	位	功能描述
Transmitter empty	[2]	当发送缓存寄存器中没有有效值且发送移位寄存器空时，则自动置为 1 0=非空　1=发送器空(发送缓存和移位寄存器)
Transmit buffer empty	[1]	当发送缓存寄存器为空，则自动置为 1 0=发送缓存寄存器不为空　1=发送缓存寄存器为空 (在非 FIFO 模式下，中断或 DMA 被请求。在 FIFO 模式下当将发送 FIFO 触发等级设为 00(空)时，中断或 DMA 被请求) 如果 UART 使用 FIFO，用户应该检查寄存器 UFSTAT 中的 Tx FIFO Count 位和 Tx FIFO Full 位而不是对此位进行检查
Receive buffer data ready	[0]	只要接收缓存寄存器保留通过 RXDn 端口接收的有效值，则自动置 1 0=缓存寄存器为空　1=缓存寄存器接收到数据(在非 FIFO 模式下，请求中断或 DMA) 如果 UART 使用 FIFO，用户应该检查 UFSTAT 中的 Rx FIFO Count 位和 Rx FIFO Full 位而不是对此位进行检查

(6) UART 错误状态寄存器(UERSTATn)

UERSTATn 共有 3 个，每个 UART 接口通道对应一个，端口地址分别为 0x50000014、0x50004014、0x50008014，只读，初始值为 0x00，各位的定义如表 4-62 所示。

表 4-62 UART 错误状态寄存器各位的定义

UERSTATn	位	功能描述
Break Detect	[3]	是否收到中止信号 0=无中止信号　1=收到中止信号(已请求中断)
Frame Error	[2]	接收操作中是否出现帧错误 0=接收过程中无帧错误　1=帧错误(已请求中断)
Parity Error	[1]	接收操作中是否出现奇偶校验错误 0=接收过程中无奇偶校验错误　1=奇偶校验错误(已请求中断)
Overrun Error	[0]	接收过程中是否出现溢出错误 0=接收过程中无溢出错误　1=溢出错误(已请求中断)

(7) UART FIFO 状态寄存器(UFSTATn)

UFSTATn 共有 3 个，每个 UART 接口通道对应一个，端口地址分别为 0x50000018、0x50004018、0x50008018，只读，初始值为 0x00，各位的定义如表 4-63 所示。

表 4-63 UART FIFO 状态寄存器各位的定义

UFSTATn	位	功能描述
Reserved	[15]	保留
Tx FIFO Full	[14]	发送操作中发送 FIFO 是否满 0=0B≤Tx FIFO data≤63B　1=Full
Tx FIFO Count	[13:8]	发送 FIFO 中的数据数量
Reserved	[7]	保留
Rx FIFO Full	[6]	接收操作中接收 FIFO 是否满 0=0B≤Rx FIFO data≤63B　1=Full
Rx FIFO Count	[5:0]	接收 FIFO 中的数据数量

(8) UART MODEM 状态寄存器(UMSTATn)

UMSTATn 共有两个,对应 UART 接口通道 0 和通道 1,端口地址分别为 0x5000001C、0x5000401C,只读,初始值为 0x00,各位的定义如表 4-64 所示。

表 4-64 UART MODEM 状态寄存器各位的定义

UMSTATn	位	功 能 描 述
Delta CTS	[4]	指出当前的 nCTS 状态和 CPU 最近一次读取的是否一致 0=未变 1=已变
Reserved	[3:1]	保留
Clear to Send	[0]	0=CTS 信号未激活(nCTS 为高电平) 1=CTS 信号激活(nCTS 为低电平)

(9) UART 发送缓冲寄存器(UTXHn)

CPU 将数据写入 UTXHn 寄存器,UART 再将数据保存到缓冲区中,并自动发送出去。在实际应用中,发送数据一般采用 DMA 的方式,或者设置发送 FIFO 为空;而接收数据时,可以设置 FIFO 为比较合适的大小。而且在接收数据时还有一个超时接收中断,可以用于接收那些由于没到达触发点而无法触发接收中断的字符。当 FIFO 中的数据量没有达到接收 FIFO 的触发等级且在 3 个字的时间内没有接收到任何数据时,产生中断,这是一种接收中断,该时间根据字长位设置。

UTXHn 共有 3 个,每个 UART 接口通道对应一个,在小端模式下端口地址分别为 0x50000020、0x50004020、0x50008020,在大端模式下端口地址分别为 0x50000023、0x50004023、0x50008023,只写,初始值未定义,各位的定义如表 4-65 所示。

表 4-65 UART 发送缓冲寄存器各位的定义

UTXHn	位	功 能 描 述
TXDATAn	[7:0]	UARTn 的发送数据

(10) UART 接收缓冲寄存器(URXHn)

URXHn 共有 3 个,每个 UART 接口通道对应一个,在小端模式下端口地址分别为 0x50000024、0x50004024、0x50008024,在大端模式下端口地址分别为 0x50000027、0x50004027、0x50008027,只读,初始值未定义,各位的定义如表 4-66 所示。

表 4-66 UART 接收缓冲寄存器各位的定义

URXHn	位	功 能 描 述
RXDATAn	[7:0]	UARTn 的接收数据

(11) UART 波特率除数寄存器(UBRDIVn)

UBRDIVn 共有 3 个,每个 UART 接口通道对应一个,端口地址分别为 0x50000028、0x50004028、0x50008028,可读写,初始值未定义,各位的定义如表 4-67 所示。

表 4-67 UART 波特率除数寄存器各位的定义

UBRDIVn	位	功 能 描 述
UBRDIVn	[15:0]	波特率分频值 UBRDIVn>0 使用 UEXTCLK 作为输入时钟,UBRDIVn 可以置 0

5. 串口编程实例

下面通过一个实例介绍如何进行串口工作参数的配置和控制数据的收发。

以下的示例将实现通过串行口 UART0 发送数据的功能，接收功能的编程与之类似。该示例的通信协议为：19200 波特率、8 位数据位、1 位停止位、无校验位。要按照 RS-232C 标准进行通信需要对 UART 的内部寄存器进行初始化。初始化要完成的工作主要有：设置数据帧格式、波特率及是否开放中断等。

下面是一个初始化 UART0 的函数。该函数被设计成通用的串口初始化函数，对 UART1、UART2 初始化的代码与此类似，只是涉及的寄存器不同而已。

在初始化 UART0 的函数中，com 参数用来确定操作的是哪个 UART 部件；data 为要发送的数据；baud 为波特率。

限于篇幅，本例只给出部分代码。函数中涉及的寄存器定义如下：

```
/*****************************************************************
 * Description: 与 UART0 初始化有关的寄存器定义
 *****************************************************************/
#define rINTMSK (*(volatile unsigned *)0x4a000008)      //中断屏蔽寄存器
#define rULCON0 (*(volatile unsigned *)0x50000000)      //UART0 线控制寄存器
#define rUCON0 (*(volatile unsigned *)0x50000004)       //UART0 控制寄存器
#define rUFCON0 (*(volatile unsigned *)0x50000008)      //UART0 FIFO 控制寄存器
#define rUMCON0 (*(volatile unsigned *)0x5000000c)      //UART0 Modem 控制寄存器
#define rUTRSTAT0 (*(volatile unsigned *)0x50000010)    //UART0 发送接收状态寄存器
#define rUERSTAT0 (*(volatile unsigned *)0x50000014)    //UART0 接收错误状态寄存器
#define rUFSTAT0 (*(volatile unsigned *)0x50000018)     //UART0 FIFO 状态寄存器
#define rUMSTAT0 (*(volatile unsigned *)0x5000001c)     //UART0 Modem 状态寄存器
#define rUBRDIV0 (*(volatile unsigned *)0x50000028)     //UART0 波特率除数寄存器
#define rUTXH0     (*(volatile unsigned char *)0x50000020)
                                                        //UART0 发送缓冲寄存器
#define rURXH0     (*(volatile unsigned char *)0x50000024)
                                                        //UART0 接收缓冲寄存器
#define pISR_UART0 (*(unsigned *)(0x33ffff90))          //中断向量指针
/*****************************************************************
 * Description:UART0 的寄存器初始化
 *****************************************************************/
void InitUART(short unsigned int com,int baud)
{
if(com==0)
{
    rUFCON0=0x0;                            //UART0 FIFO 控制寄存器,FIFO 禁止
    rULCON0=(0<<6)|(0<<3)|(0<<2)|(3);
                                            //正常模式,无奇偶校验,1 位停止位,8 位数据位
    rUCON0=0x245;                           //UART0 控制寄存器
    rUBRDIV0=((int)(Pclk/16/baud+0.5)-1);            //波特率除数寄存器,Pclk 为常量
    rINTMSK=0xeffffff;                               //开启中断允许功能
    pISR_UART0=(unsigned)Uart0_RxInt;
    }
else
{
…                                                    //UART1、UART2 的初始化
```

```
}}
/*****************************************************************
 * Description:UART0 发送一个字符
*****************************************************************/
void Uart0_TxByte(short unsigned int com,short unsigned int data)
{
    if(com==0)
    {
While((rUTRSTAT0&0x4)!=0x4);                         //等待发送完成
rUTXH0=data;
    }
else
{
…                                                    //UART1、UART2 的发送程序
}
}
/*****************************************************************
 * Description:UART0 接收一个字符
*****************************************************************/
void __irq Uart0_RxInt(void)
{
    short unsigned int data;
While((rUTRSTAT0&0x1)==0x1);                         //若接收到有效数据
data=rURXH0;
…                                                    //实际需要的其他功能语句
}
rSRCPND=0x10000000;                                  //清除中断未决位
}
```

通信完成后，通常需要关闭 RS-232 串口。可以使用下面的程序段关闭 UART0：

```
/*****************************************************************
 * Description:关闭 UART0
*****************************************************************/
void rs232_Close(short unsigned int com)
{
if(com==0)
{
rINTMSK=rINTMSK|(0x10000000);
}
else
{
…                                                    //UART1、UART2 的关中断代码
}
}
```

4.3.6　定时器

任务：掌握 S3C2440A 的定时器的功能、结构及初始化。

S3C2440A 具有 5 个 16 位定时器。定时器 0、1、2、3 具有 PWM 功能，可以在中断模式或 DMA 模式下工作。定时器 4 是一个内部定时器，不具有对外输出口线，不能实现 PWM 功能。定时器 0 还具有死区发生器，通常用于大电流设备。

1. 定时器内部结构

一般来说，微处理器的主时钟源主要是外部晶振或外部时钟，而用得最多的是外部晶振。由于外部时钟源的频率一般不能满足系统所需要的高频条件，所以往往需要 PLL（锁相环）进行倍频处理。在 S3C2440A 中，有两个不同的 PLL，一个是 MPLL，另一个是 UPLL。UPLL 用于提供 48MHz 的 USB 时钟。其中，外部时钟源经过 MPLL 处理后能够得到 3 个不同的系统时钟：FCLK、HCLK 和 PCLK。FCLK 是主频时钟，用于 ARM920T 内核；HCLK 用于 ARM920T 的内存控制器、中断控制器、LCD 控制器、DMA 控制器以及 USB 主模块等快速部件；PCLK 用于看门狗、IIS、IIC、PWM、MMC 接口、ADC、UART、GPIO、RTC 以及SPI 等速度较慢的部件。

定时器的时钟源是 PCLK，定时器内部有两个 8 位预分频器和两个具有 5 种分频系数（1/2、1/4、1/8、1/16 和 TCLK）的 4 位时钟分割器。定时器 0 和定时器 1 共用一个 8 位预分频器、一个 4 位时钟分割器；定时器 2、定时器 3、定时器 4 共用另一个 8 位预分频器和另一个 4 位时钟分割器。8 位预分频器和 4 位时钟分割器均可编程设定。S3C2440A 的定时器内部结构如图 4-12 所示。

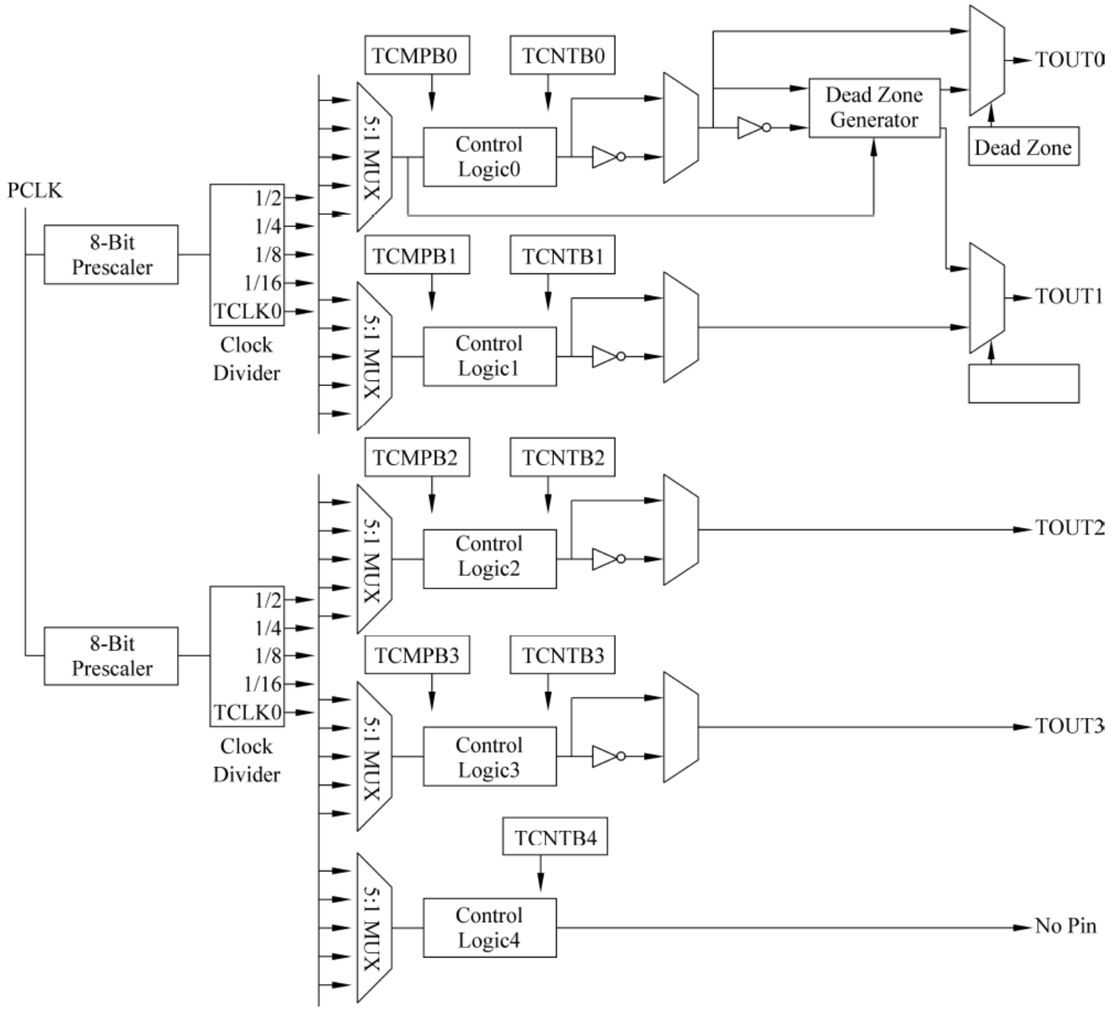


图 4-12 定时器内部结构图

时钟源 PCLK 输入后，首先经过预分频器降低频率后，进入第 2 个分频，可以生成 5 种不同的分频信号。其中 TCLK0、TCLK1 是 S3C2440A 的外部时钟信号输入引脚。

每个时钟分割器从与之对应的 8 位预分频器得到时钟源，每个定时器从时钟分割器的输出得到各自的时钟源。

8 位预分频器是可编程选择的，它的频率由 PCLK 除以保存在定时器配置寄存器 TCFG0 中的除数的结果设定。某个定时器取时钟分割器的哪一个分频信号作为时钟源是由定时器配置寄存器 TCFG1 中的 MUXn 的 4 位决定的。

2. 基本定时器操作

所有定时器都是递减计数。

除了定时器 4 外，每个定时器具有一个倒计时器，实际上就是一个通过定时器时钟源驱动的 16 位递减计数寄存器 TCNTn。当递减计数寄存器值减到 0 时，定时器中断请求就产生了，这个中断用于通知 CPU 定时器定时已经完成。

除了定时器 4 外，每个定时器还有计数缓冲寄存器 TCNTBn、计数比较缓冲寄存器 TCMPBn 和计数比较寄存器 TCMPn。TCNTn 中的值是定时器当前的计数值，而 TCNTBn 用来在计数开始时为 TCNTn 赋计数初值；TCMPn 和 TCMPBn 用于脉宽调制。当 TCNTn 的值和 TCMPn 中的值相等时，定时器控制逻辑将改变输出电平，TCMPBn 用来为 TCMPn 赋比较初值。

定时器有两种操作模式：单次触发模式和自动重载模式。

在单次触发模式下，定时器完成一次递减计数并产生中断请求后，定时器便停止了。如果要启动下一次定时，需要重新向 TCNTBn 中写入计数值，并重新启动定时器开始工作。

在自动重载模式下，当定时器递减计数减到 0 时，TCNTBn 的值就会被自动载入到 TCNTn 中继续开始下一次定时。但是，如果定时器停止(例如，在运行中清除了定时器控制寄存器 TCONn 中的定时器使能位)，那么 TCNTBn 的值就不会被重新载入到 TCNTn 中，定时器便停止下来。

3. 自动重载和双缓冲器

双缓冲器是指定时器计数缓冲寄存器 TCNTBn 和定时器比较缓冲寄存器 TCMPBn，它们都分别具有一个初始值，用来为计数寄存器 TCNTn 和比较寄存器 TCMPn 赋值。TCNTBn 和 TCMPBn 这两个缓冲寄存器的应用使得用户能够在定时器运行期间修改运行参数，而不用担心影响定时器当前的工作状态，即能够使定时器在频率和占空比被更改时仍产生一个稳定输出。

若 TCON 中定时器 n 的自动重载模式开启，则定时器工作在自动重载模式下，TCNTBn 和 TCMPBn 的值将在定时器的计数值达到 0 时分别载入到 TCNTn 和 TCMPn 中。

定时器计数值可以直接写入 TCNTBn，但不能直接写入 TCNTn；而当前定时器的计数值，即 TCNTn 的值，可以通过定时器计数观察寄存器 TCNTOn 来读取。如果读取 TCNTBn，那么读出的数值不一定是当前定时器的计数值，但一定是下一个定时周期的计

数值。

4. 设置手动更新位和反转器开关位初始化定时器

当 TCNTn 减到 0 时,定时器的自动重载操作就会发生,但在第 1 次重载发生之前 TCNTn 的初始值还未定义过。在这种情况下,要通过设置手动更新位的方法向 TCNTn 中加载 TCNTBn 中的初始值。同样,如果定时器被强制停止,TCNTn 中保存着某一大小的计数值,可能并不是从 TCNTBn 中重新载入的初值。当重新启动定时,需要将 TCNTn 设置为新值时,就需要采用手动更新的方式。同时反转器开关状态的改变将直接导致定时器输出 TOUTn 的逻辑电平改变,因此建议在启动定时器之前,配置好反转器的开关位。

手动更新位和反转器开关位都位于定时器控制寄存器 TCON 中,因此,通过对 TCON 寄存器写入相应的值来初始化和启动定时器。

启动定时器的一般步骤如下。

(1) 将计数器初始值写入到 TCNTBn 和 TCMPBn 中。

(2) 在 TCON 中,设置对应定时器的手动更新位。建议同时设置对应反转器的开关位。

(3) 在 TCON 中,设置对应定时器的启动位来启动定时器(同时,清除手动更新位)。

关于具体的设置操作及结果在 S3C2440A 的数据手册里给出了一个例子,很好地说明了定时器的工作过程,如图 4-13 所示,具体步骤如下。

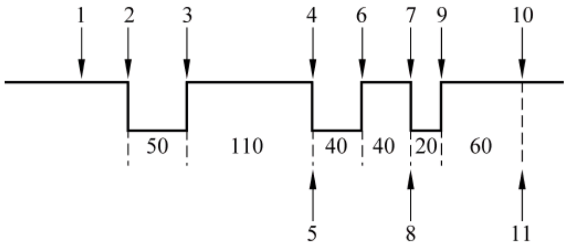


图 4-13 定时器操作示例

(1) 先启用 TCON 寄存器的自动重载功能,然后设置 TCNTBn 为 160(50+110),TCMPBn 为 110,接着设置 TCON 中的手动更新位和反转器位(ON/OFF)。手动更新的设置使得 TCNTn 和 TCMPn 分别载入了 TCNTBn 和 TCMPBn 的值。最后,分别设置 TCNTBn 和 TCMPBn 为 80(40+40)和 40,作为下一个周期的重置值。

(2) 设置 TCON 中的启动位,同时将手动更新位清 0,反转器置 OFF,自动重载使能。定时器的递减计数器被启动并开始工作。

(3) 当 TCNTn 具有与 TCMPn 相同的值时,TOUTn 的逻辑电平从低变高。

(4) 当 TCNTn 到达 0 时,引发中断请求,TOUTn 的逻辑电平从高变低,同时 TCNTBn 的值载入到一个临时寄存器中。在下一个定时器节拍,TCNTn 从临时寄存器重新载入计数值。

(5) 在 ISR(中断服务程序)中,分别将 TCNTBn 和 TCMPBn 设置为 80(20+60)和 60,用于下一周期。

(6) 当 TCNTn 具有与 TCMPn 相同的值时，TOUTn 的逻辑电平从低变高。

(7) 当 TCNTn 到达 0 时，TCNTn 自动重新载入 TCNTBn 的值。同时，引发中断请求，TOUTn 的逻辑电平从高变低。

(8) 在 ISR 中，禁用 TCON 的自动重载功能，同时禁止该定时器的中断请求，从而停止定时器的工作。

(9) 当 TCNTn 具有与 TCMPn 相同的值时，TOUTn 的逻辑电平从低变高。

(10) 当 TCNTn 减到 0 时，由于自动重载功能被禁用了，因此 TCNTn 不再重载计数值，且定时器也停止工作了。

(11) 不再产生中断请求。

5. PWM(脉宽调制)

PWM 脉冲频率由 TCNTBn 决定。PWM 脉冲宽度值则由 TCMPBn 的值来决定：如果要得到一个较小的 PWM 脉宽输出值，就可以减小 TCMPBn 的值；反之，则增大 TCMPBn 的值。如果输出反转器被使能，增大和减小的结果也将是反转的。

基于双缓冲器的特性，下一个 PWM 周期的 TCMPBn 值可以通过 ISR 或其他手段在当前 PWM 周期中的任何一点写入。

6. 输出电平控制

可以用以下办法来保持 TOUT 为高或低(假设反转器为 OFF)。

(1) 禁用自动重载位后 TOUTn 变为高电平，定时器在 TCNTn 减到 0 时停止。推荐采用这个模式。

(2) 通过将定时器的启动/停止位清 0 来停止定时器。如果 TCNTn≤TCMPn，输出电平为高；如果 TCNTn>TCMPn，输出电平为低。

(3) 在 TCMPBn 中写入比 TCNTBn 大的值，使得 TCMPn 与 TCNTn 的值不可能有相同的机会，也就不会引起 TOUTn 跳变，从而禁止 TOUTn 变高。

(4) 通过设置 TCONn 中反转器的 ON/OFF 位来反转 TOUTn。

7. 死区发生器

死区发生器用于对大功率设备进行 PWM 控制，这一特性用于在一个开关设备的断开和另一个开关设备的闭合之间插入一个时间缺口，用来阻止两个开关设备处于同时闭合的状态(即使是非常短的时间)。

TOUT0 是一个 PWM 输出。nTOUT0 是 TOUT0 的反转输出。如果死区被使能，TOUT0 和 nTOUT0 的输出波形将会是 TOUT0_DZ 和 nTOUT0_DZ。在死区间隔中，TOUT0_DZ 和 NTOUT0_DZ 肯定不会同时闭合。

8. 定时器寄存器

(1) 定时器配置寄存器 0(TCFG0)

TCFG0 用于设置定时器的输入时钟频率。输入时钟频率的计算公式如下：

$$输入时钟频率=PCLK/(预分频值+1)/(分割值)$$

其中，(预分频值)=0～255；(分割值)=2、4、8、16。

TCFG0 端口地址为 0x51000000，可读写，初始值为 0x0，各位的定义如表 4-68 所示。

表 4-68 定时器配置寄存器 0 各位的定义

TCFG0	位	功 能 描 述
Reserved	[31:24]	保留
Dead zone length	[23:16]	确定死区长度 死区长度的 1 个单位定时器 0 的定时间隔
Prescaler1	[15:8]	确定定时器 2、3、4 的预分频器的值
Prescaler0	[7:0]	确定定时器 0、1 的预分频器的值

(2) 定时器配置寄存器 1(TCFG1)

TCFG1 用于选择定时器的 DMA 模式及多路选择器,端口地址为 0x51000004,可读写,初始值为 0x0,各位的定义如表 4-69 所示。

表 4-69 定时器配置寄存器 1 各位的定义

TCFG1	位	功 能 描 述
Reserved	[31:24]	保留
DMA mode	[23:20]	选择产生 DMA 请求的定时器 0000=不选择 DMA 方式(所有定时器采用中断方式) 0001=Timer0 0010=Timer1 0011=Timer2 0100=Timer3 0101=Timer4 0110=保留
MUX4	[19:16]	选择 Timer4 的分割器值 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01xx=外部 TCLK1
MUX3	[15:12]	选择 Timer3 的分割器值 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01xx=外部 TCLK1
MUX2	[11:8]	选择 Timer2 的分割器值 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01xx=外部 TCLK1
MUX1	[7:4]	选择 Timer1 的分割器值 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01xx=外部 TCLK0
MUX0	[3:0]	选择 Timer0 的分割器值 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01xx=外部 TCLK0

(3) 定时器控制寄存器(TCON)

TCON 可进行定时器的自动重载、手动更新、启动/停止、输入反转及死区使能的设置,端口地址为 0x51000008,可读写,初始值为 0x0,各位的定义如表 4-70 所示。

表 4-70 定时器控制寄存器各位的定义

TCON	位	功 能 描 述
Timer4 auto reload on/off	[22]	确定 Timer4 的自动重载功能位 0=单次触发模式 1=自动重载模式
Timer4 manual update(note)	[21]	确定 Timer4 的手动更新位 0=不操作 1=更新 TCNTB4

续表

TCON	位	功 能 描 述
Timer4 start/stop	[20]	确定 Timer4 的启动/停止位 0=停止　1=启动
Timer3 auto reload on/off	[19]	确定 Timer3 的自动重载功能位 0=单次触发模式　1=自动重载模式
Timer3 output inverter on/off	[18]	确定 Timer3 的输出反转位 0=TOUT3 不反转　1=TOUT3 反转
Timer3 manual update (note)	[17]	确定 Timer3 的手动更新位 0=不操作　1=更新 TCNTB3 和 TCMPB3
Timer3 start/stop	[16]	确定 Timer3 的启动/停止位 0=停止　1=启动
Timer2 auto reload on/off	[15]	确定 Timer2 的自动重载功能位 0=单次触发模式　1=自动重载模式
Timer2 output inverter on/off	[14]	确定 Timer2 的输出反转位 0=TOUT2 不反转　1=TOUT2 反转
Timer2 manual update (note)	[13]	确定 Timer2 的手动更新位 0=不操作　1=更新 TCNTB3 和 TCMPB3
Timer2 start/stop	[12]	确定 Timer2 的启动/停止位 0=停止　1=启动
Timer1 auto reload on/off	[11]	确定 Timer1 的自动重载功能位 0=单次触发模式　1=自动重载模式
Timer1 output inverter on/off	[10]	确定 Timer1 的输出反转位 0=TOUT1 不反转　1=TOUT1 反转
Timer1 manual update (note)	[9]	确定 Timer1 的手动更新位 0=不操作　1=更新 TCNTB3 和 TCMPB3
Timer1 start/stop	[8]	确定 Timer1 的启动/停止位 0=停止　1=启动
Reserved	[7:5]	保留
Dead zone enable	[4]	确定死区操作位 0=禁止　1=允许
Timer0 auto reload on/off	[3]	确定 Timer0 的自动重载功能位 0=单次触发模式　1=自动重载模式
Timer0 output inverter on/off	[2]	确定 Timer0 的输出反转位 0=TOUT0 不反转　1=TOUT0 反转
Timer0 manual update (note)	[1]	确定 Timer0 的手动更新位 0=不操作　1=更新 TCNTB3 和 TCMPB3
Timer0 start/stop	[0]	确定 Timer0 的启动/停止位 0=停止　1=启动

(4) 计数缓冲寄存器(TCNTBn)

TCNTBn 共有 5 个,每个 timer 对应一个,端口地址分别为 0x5100000C、0x51000018、0x51000024、0x51000030、0x5100003C,可读写,初始值为 0x0,各位的定义如表 4-71 所示。

表 4-71　计数缓冲寄存器各位的定义

TCNTBn	位	功 能 描 述
Timern count buffer register	[15:0]	为定时器 n 设置计数缓冲值

(5) 比较缓冲寄存器(TCMPBn)

TCMPBn 共有 4 个，timer0～timer3 各对应一个，端口地址分别为 0x51000010、0x5100001C、0x51000028、0x51000034，可读写，初始值为 0x0，各位的定义如表 4-72 所示。

表 4-72 比较缓冲寄存器各位的定义

TCMPBn	位	功 能 描 述
Timern compare buffer register	[15:0]	为定时器 n 设置比较缓冲值

(6) 计数观察寄存器(TCNTOn)

TCNTOn 共有 5 个，每个 timer 对应一个，端口地址分别为 0x51000014、0x51000020、0x5100002C、0x51000038、0x51000040，只读，初始值为 0x0，各位的定义如表 4-73 所示。

表 4-73 计数观察寄存器各位的定义

TCNTOn	位	功 能 描 述
Timern observation register	[15:0]	给出定时器 n 当前的计数值

9. 定时器应用编程

定时器的应用非常广泛，也非常灵活，对于不同的应用要求，定时器初始化编程也不同。但无论用户要求有什么不同，均应根据用户要求，计算需要的定时间隔、脉宽等参数，进而确定预分频系数、分割器值，写入对应的寄存器。

例如，若需要产生一个周期约为 200ms 的脉冲信号，系统的 PCLK 为 66MHz，选用 Timer0 来产生该脉冲信号。

首先，确定预分频系数、分割器值。预分频系数可在 0～255 间选择，分割器值则在 2、4、8、16 间选择，选择时要将二者结合起来，总的原则就是计算出来的计数常数不能超过 16 位寄存器能表示的计数范围，即计数常数应在 0～65535 范围内。本例预分频系数选择 32，分割器值选择 16，则计数常数为：

$$\text{计数常数} = \text{定时时间间隔}/(1/(\text{PCLK}/(\text{预分频系数}+1)/(\text{分割器值})))$$
$$= 200\text{ms}/(1/(66\text{M}/33/16)) = 25000$$

计算出计数常数后，即可进行定时器初始化。在初始化程序中，首先设置 TCFG0 和 TCFG1 寄存器进行预分频系数、分割器值选择，然后再将计数常数写入 TCNTB0，最后通过设置 TCON 寄存器启动 Timer0。Timer0 启动后，当其递减计数器的值减到 0 时，会在 TOUT0 引脚产生“回 0 信号”，该信号即是符合要求的脉冲信号。计数到 0 时也会产生中断请求信号。若需要由微处理器处理中断，则还需要初始化中断控制寄存器，这里只给出初始化程序段。初始化程序如下：

```
;*****************************************************************
;Description:    用 Timer0 实现定时
;*****************************************************************
#define rTCFG0(* (volatile unsigned* )0x51000000)           //定时器配置寄存器 0
#define rTCFG1(* (volatile unsigned* )0x51000004)           //定时器配置寄存器 1
#define rTCON(* (volatile unsigned* )0x51000008)            //定时器控制寄存器
#define rTCNTB0(* (volatile unsigned* )0x5100000c)          //Timer0 计数缓冲寄存器
void TestTimerInit(void)
```

```
{
    rTCFG0=0x1f;                //TCFG0 设置,Dead zone length=0,Timer0 预分频系数=32
    rTCFG1=0x03;                //TCFG1 设置,工作在中断方式下,MUX0=1/16
    rTCNTB0=25000;              //设置计数常数
    rTCON=0x02;                 //更新 TCNTB0 和 TCMPB0
    rTCON=0x09;                 //设置 Timer0 自动装载,并启动
    …                           //其他程序语句
}
```

4.4　嵌入式系统硬件基本电路

问题：实现嵌入式系统硬件设计需要哪些硬件电路？各需要考虑哪些方面？
重点：嵌入式系统硬件基本电路。
内容：嵌入式系统硬件基本电路介绍，包括电源、复位、晶振电路、存储器接口和 JTAG 接口、串行接口等。

嵌入式系统的硬件是嵌入式系统软件环境运行的基础，它提供了软件运行的物理平台和通信接口。嵌入式系统硬件基本电路包括电源、复位、晶振电路、存储器接口和 JTAG 接口。

1．电源电路

电源电路是整个系统正常工作的基础，电源电路必须满足系统对电路性能指标的要求。ARM 构成的系统的电源通常不是一组，每一组的电源也不一定一样。如果在某个部位该供电时没有供电，也只有那一部位工作会不正常，并不会影响到整体工作，这是 ARM 构成的系统的优点，也是它的缺点。优点是每一部分不会互相影响，缺点是用户以为系统已经正常运作，但它却潜藏了不可预知的错误。通常电源的电压会根据不同的系统而有所不同，可分为以下几组。

(1) 由 ARM CPU 核使用。
(2) 由系统的 PLL 使用。
(3) 由系统的 GPIO 使用。
(4) 由系统的各种控制器使用，根据不同的控制器也可分为多组电源输入。

之所以分成很多不同的电源输入，主要是为了省电。在实际设计中可根据需要设计电路，关掉不需要的电源。一般的电源电路设计模块如图 4-14 所示。

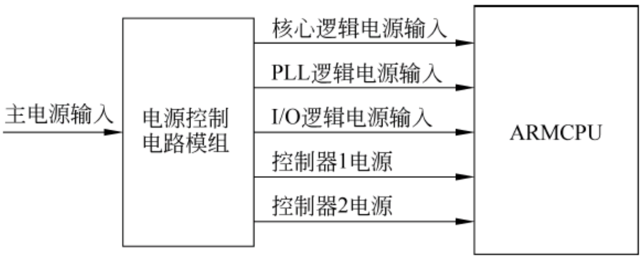


图 4-14　ARM 电源模块示意图

2. 晶振电路

晶振电路用于为微处理器及其他电路提供工作时钟，是系统必需的重要电路。大部分的 ARM 片上系统都会提供两种不同的时钟生成方法：晶振（石英晶体振荡器）和外部时钟，同时会用硬件来设置使用哪种方法生成时钟，不同方法的接线和引脚也不一样。典型晶振电路如图 4-15 所示。

3. 复位电路

复位电路主要完成系统的上电复位和系统在运行时用户的按键复位功能，可由简单的 RC 电路构成，也可使用其他的相对较复杂，但功能更完善的电路。如采用专用复位芯片进行复位，稳定可靠。较简单的 RC 复位电路如图 4-16 所示。

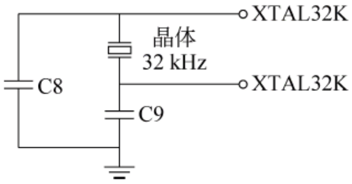


图 4-15 晶振电路

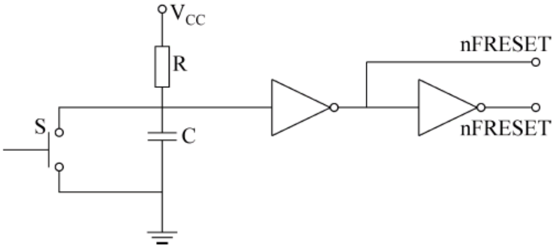


图 4-16 RC 复位电路

4. SDRAM 接口电路

SDRAM 具有单位空间存储容量大和价格便宜的优点，已广泛应用在各种嵌入式系统中。SDRAM 在系统中主要用做程序的运行空间、数据及堆栈区。因此，SDRAM 接口电路在最小系统设计中必须高度重视。

SDRAM 的存储单元可以理解为一个电容，总是倾向于放电，为避免数据丢失，必须定时刷新（充电）。因此，要在系统中使用 SDRAM，就要求微处理器具有刷新控制逻辑，或在系统中另外加入刷新控制逻辑电路。S3C2440A 及其他一些 ARM 芯片在片内具有独立的 SDRAM 刷新控制逻辑，可方便地与 SDRAM 接口。但某些 ARM 芯片则没有 SDRAM 刷新控制逻辑，就不能直接与 SDRAM 接口，在进行系统设计时应注意这一点。

目前常用的 SDRAM 的数据宽度为 8 位/16 位，工作电压一般为 3.3V，主要生产厂商为 Samsung、HYUNDAI、Winbond 等，若同类器件具有相同的电气特性和封装形式，则可通用。

根据系统的需求，可构建 16 位或 32 位的 SDRAM 存储器系统，这里采用两片 HY57V561620 并联构建 32 位的 SDRAM 存储器系统，单片 HY57V561620 的数据宽度为 16 位，容量为 32MB，用两片 HY57V561620 构建 64MB 的 SDRAM 空间，可满足嵌入式操作系统及各种较复杂的运行需求。两片 HY57V561620 构建 32 位 SDRAM 空间的接口电路如图 4-17 所示。

两片 HY57V561620，其中一片为高 16 位，另一片为低 16 位。将 S3C2440A 的 nSCS0 接至两片 HY57V561620 的 nCS 端，两片 HY57V561620 的 CLK 端接 S3C2440A 的 SCLK0 端，两片 HY57V561620 的 CKE 端接 S3C2440A 的 SCKE 端；两片 HY57V561620 的 nRAS、nSCAS、nSWE 端分别接 S3C2440A 的 nSRAS、nSCAS、nWE 端；两片 HY57V561620 的地址总线 A[12:0]接 S3C2440A 的 A[14:2]；两片 HY57V561620 的 BA

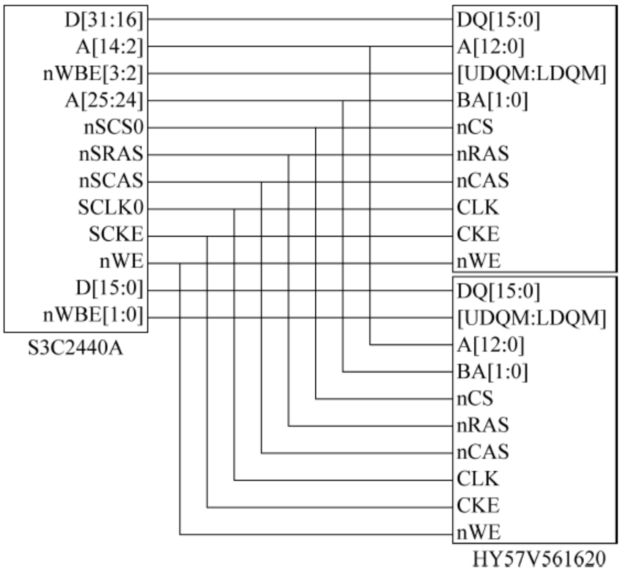


图4-17　两片 HY57V561620 构建 32 位 SDRAM 空间的接口电路

[1:0]接 S3C2440A 的地址总线 A[25:24];高 16 位片的 DQ[15:0]接 S3C2440A 的数据总线 D[31:16],低 16 位片的 DQ[15:0]接 S3C2440A 的数据总线 D[15:0];高 16 位片的[UDQM:LDQM]接 S3C2440A 的 nWBE[3:2],低 16 位片的[UDQM:LDQM]接 S3C2440A 的 nWBE[1:0]。

5. 串行接口电路

S3C2440A 提供了串行接口,使用 RS-232 标准接口,在近距离通信系统中可直接进行端对端的连接,但由于 S3C2440A 系统中 LVTTL 电路的逻辑电平与 RS-232 标准逻辑电平不相匹配,二者间要进行正常的通信,必须经过信号电平转换,可使用电平转换芯片进行电平转换,如 MAX3221。以 RS-232 标准 9 芯 D 型接口、MAX3221 转换芯片为例,要完成最基本的串行通信功能,只需要正确连接 RxD、TxD 和 GND(地)端即可。串行接口电路如图 4-18 所示。

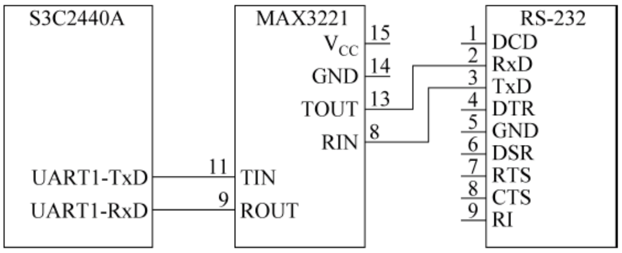


图 4-18　串行接口电路

6. Nand Flash 存储器接口电路

Flash 存储器是一种可在系统(In-System)进行电擦写、掉电后信息不丢失的存储器。它具有功耗低、容量大、擦写速度快、可整片或分扇区在系统编程(烧写)和擦除等特点,并且可由内部嵌入的算法完成对芯片的操作,因而在各种嵌入式系统中得到了广泛的应用。作为一种非易失性存储器,Flash 在系统中通常用于存放程序代码、常量表以及一些在系统掉

电后需要保存的用户数据等。常用的 Flash 为 8 位或 16 位数据宽度，编程电压为单 3.3V。主要生产厂商为 Intel、Atmel、Hyundai 等，生产的同型器件一般具有相同的电气特性和封装形式，可根据需要选用。

这里以 1 片 K9F6408 为例，构建 8 位 Flash 存储器系统的 Nand Flash 接口电路如图 4-19 所示。

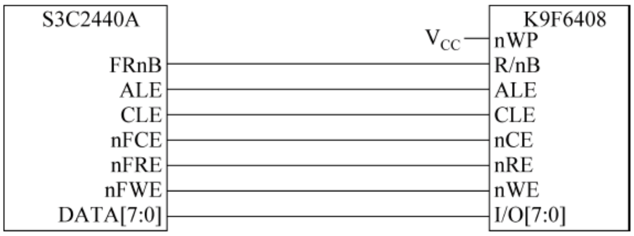


图 4-19 1 片 K9F6408 组成 8 位 Flash 存储器系统的接口电路

7. JTAG 接口电路

JTAG 技术是一种嵌入式调试技术，芯片内部封装了专门的测试电路 TAP（Test Access Port 测试访问口），通过专用的 JTAG 测试工具对内部节点进行测试和控制，目前大多数 ARM 器件都支持 JTAG 协议，标准 JTAG 接口是 4 线：TMS（测试模式选择）、TCK（测试时钟）、TDI（测试数据串行输入）、TDO（测试数据串行输出）。14 针 JTAG 接口与 S3C2440A 的连接电路如图 4-20 所示。

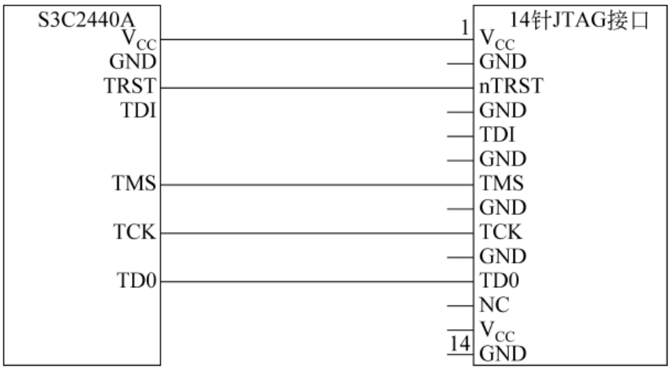


图 4-20 JTAG 接口与 S3C2440A 的连接电路

4.5 S3C2440A 启动程序

问题： S3C2440A 内包含了各种外围部件，如何在系统启动后正确地驱动这些部件？
重点： 在用户的应用程序运行之前，对系统的初始化。
内容： 用汇编语言设置各种外围部件需要的工作状态。

在嵌入式系统启动的过程中，往往需要对多数硬件模块进行配置和准备必要的运行环境，还需要执行程序来设置工作模式，因此在用户的应用程序之前，需要使用一段专门的代码来完成对系统的初始化。由于这类代码直接面向微处理器内核和硬件控制器进行编程，

一般都使用汇编语言。一般包括如下通用的内容。

1. 中断向量表

ARM 要求中断向量表必须放置在从 0 地址开始，连续 8×4 字节的空间内。

每当一个中断发生以后，ARM 处理器便强制把 PC 指针置为向量表中对应中断类型的地址值。因为每个中断向量占据向量表中 1 个字的存储空间，只能放置一条 ARM 指令，用来使程序跳转到存储器的其他地方，再执行中断处理。

中断向量表的程序实现通常如下所示：

```
AREA Init,CODE,READONLY
ENTRY
B     ResetHandler
B     UndefHandler
B     SWIHandler
B     PreAbortHandler
B     DataAbortHandler
B     .
B     IRQHandler
B     FIQHandler
```

其中，关键字 ENTRY 用于指定编译器保留这段代码，因为编译器可能会认为这是一段冗余代码而加以优化。链接时要确保这段代码被链接在 0 地址处，并且作为整个程序的入口。

2. 初始化存储器系统

通常 Flash 和 SRAM 同属于静态存储器类型，可以合用同一个存储器端口；而 DRAM 因为有动态刷新和地址线复用等特性，通常配有专用的存储器端口。

存储器端口的接口时序优化是非常重要的，这会影响到整个系统的性能。因为一般系统运行速度的瓶颈都在于对存储器的访问，所以存储器访问时序应尽可能快，而同时又要考虑到由此带来的稳定性问题。

初始化存储器系统时还需要考虑存储器地址分布。S3C2440A 确定了 SDRAM 对应的 bank 地址是 0x30000000。

3. 初始化堆栈

因为 ARM 有 7 种工作模式，每一种模式的堆栈指针寄存器(SP)都是独立的。因此，对程序中需要用到的每一种模式都要为 SP 定义初值：先改变 CPSR 内的状态位，使处理器切换到不同的状态，然后给 SP 赋值。注意，不要切换到 User 模式进行 User 模式的堆栈设置，因为进入 User 模式后就不能再操作 CPSR 回到其他模式了，可能会对接下去的程序执行造成影响。

这是一段堆栈初始化的代码示例，其中只定义了 3 种模式的 SP 指针：

```
FIQMODE     EQU     0x11                    ;工作模式常量定义
IRQMODE     EQU     0x12
SVCMODE     EQU     0x13
MODEMASK    EQU     0x1f
NOINT       EQU     0xc0
SVCStack    EQU     0x33ff5800              ;栈顶指针常量定义
```

```
IRQStack    EQU    0x33ff7000
FIQStack    EQU    0x33ff8000
MRS    R0,CPSR
BIC    R0,R0,#MODEMASK                 ;安全起见,屏蔽模式位以外的其他位
ORR    R1,R0,#IRQMODE|NOINT
MSR    CPSR_cxfs,R1
LDR    SP,=IRQStack                    ;IRQ模式 SP初始化
ORR    R1,R0,#FIQMODE|NOINT
MSR    CPSR_cxsf,R1
LDR    SP,=FIQStack                    ;FIQ模式 SP初始化
ORR    R1,R0,#SVCMODE|NOINT
MSR    CPSR_cxsf,R1
LDR    SP,=SVCStack                    ;SVC管理模式 SP初始化
```

4. 初始化有特殊要求的端口、设备

若对某些外围设备有特殊要求，需要进行初始化。如在系统启动之初，希望连接到 GPIO 口的几个 LED 能够顺序闪烁，则需要对对应的 GPIO 寄存器进行设置和传输信号。

以下为复位模式下的中断屏蔽设置和关闭看门狗设置：

```
WTCON        EQU    0x53000000
INTMSK       EQU    0x4A000008
INTSUBMSK    EQU    0x4A00001C
ResetHandler
    ldr    r0,=WTCON                  ;看门狗关闭
    ldr    r1,=0x0
    str    r1,[r0]
    ldr    r0,=INTMSK
    ldr    r1,=0xffffffff             ;禁止所有的中断
    str    r1,[r0]
    ldr    r0,=INTSUBMSK
    ldr    r1,=0x7fff                 ;禁止所有的子中断
    str    r1,[r0]
```

5. 初始化应用程序执行环境

所谓应用程序执行环境的初始化，就是完成必要的从 ROM 到 RAM 的数据传输和内容清 0。

在 ARM 的集成开发环境中，只读的代码段和常量被称为 RO 段(ReadOnly)；可读写的全局变量和静态变量被称为 RW 段(ReadWrite)；RW 区中要被初始化为零的变量被称为 ZI 段(ZeroInit)。

映像一开始总是存储在 ROM/Flash 里面，其 RO 部分既可以在 ROM/Flash 里面执行，也可以转移到速度更快的 RAM 中执行；而 RW 和 ZI 这两部分必须转移到可写的 RAM 里去。下面是在 ADS 下一种常用存储器模型的直接实现：

```
LDR    r0,=|Image$$RO$$Limit|         ;得到 RW数据源的起始地址
LDR    r1,=|Image$$RW$$Base|          ;RW段在 RAM里的执行段起始地址
LDR    r2,=|Image$$ZI$$Base|          ;ZI段在 RAM里的起始地址
CMP    r0,r1                          ;比较它们是否相等
BEQ    %F1
```

```
0       CMP       r1,r3
        LDRCC     r2,[r0],#4
        STRCC     r2,[r1],#4
        BCC       %B0
1       LDR       r1,=|Image$$ZI$$Limit|
        MOV       r2,#0
2       CMP       r3,r1
        STRCC     r2,[r3],#4
        BCC       %B2
```

程序实现了 RW 段数据的复制和 ZI 段的清零功能。其中引用的 4 个符号是由链接器输出的。

(1) |Image$$RO$$Limit|

表示 RO 段末地址后面的地址,即 RW 段数据源的起始地址。

(2) |Image$$RW$$Base|

RW 段在 RAM 里的执行区起始地址,也就是编译器选项 RW_Base 指定的地址。

(3) |Image$$ZI$$Base|

ZI 段在 RAM 里的起始地址。

(4) |Image$$ZI$$Limit|

ZI 段在 RAM 里的结束地址后面的一个地址。

程序先把 ROM 里|Image$$RO$$Limt|开始的 RW 初始数据复制到 RAM 里|Image$$RW$$Base|开始的地址,当 RAM 的目标地址到达|Image$$ZI$$Base|后就表示 RW 段结束和 ZI 段开始,然后对 ZI 段进行清零操作,直到遇到结束地址|Image$$ZI$$Limit|。

6. 改变处理器模式

因为在初始化过程中,许多操作需要在特权模式下才能进行(如对 CPSR 的修改),所以要特别注意不能过早地进入用户模式。

内核级的中断使能也可以考虑在这一步进行。如果系统中另外存在一个专门的中断控制器,这么做也安全。

7. 呼叫主应用程序

当所有的系统初始化工作完成之后,就需要把程序流程转入主应用程序。最简单的一种情况是直接从启动代码跳转到应用程序的主函数入口:

```
IMPORT main
B main
```

主函数名称可以由用户随便定义。

在 ARM ADS 环境中,还另外提供了一套系统级的呼叫机制:

```
IMPORT __main
B __main
```

__main 是编译系统提供的一个函数,负责完成库函数和应用程序执行环境的初始化,最后自动跳转到 main 函数。

以上这些工作都可以不通过操作系统帮助,而直接让一个系统开始操作,即构筑无操作

系统的嵌入式系统直接运行应用程序。只不过如果不通过操作系统的话,开发复杂的应用程序会非常辛苦。开发简单的系统这些知识确实已经足够,但对于较复杂一些的系统,仅仅掌握这些知识是不够的。所以接下来的章节中将会讨论有关嵌入式操作系统的内容。

小结

本章引入了常用的嵌入式系统硬件设计方法——最小系统设计法,阐述了嵌入式硬件最小系统包含的硬件及其功能。在介绍最小系统的基础上,逐步展开以嵌入式微处理器为核心的各个硬件部分的原理介绍。以 Samsung 公司基于 ARM920T 核的 S3C2440A 芯片为例,讨论了嵌入式系统设计中常用的外围接口部件,包括存储器控制器、Nand Flash 控制器、定时器等,对各外围接口部件工作原理进行了较详细的介绍,给出了各个部件的编程应用。在介绍外围接口部件的基础上还介绍了嵌入式系统硬件设计的典型基本接口电路,让读者对嵌入式系统硬件设计过程有一个较为全面的了解,便于应用。

在本章的最后给出了 S3C2440A 在没有操作系统支持下的硬件引导程序,阐述了相关硬件部件的初始化过程、必要的程序环境初始化及如何转向应用程序。

第5章

嵌入式操作系统基础

学习目标

通过本章的学习，应该掌握：

- 操作系统的基本概念
- 操作系统的功能
- 多任务之间的通信机制
- 操作系统的类型和当今流行的嵌入式操作系统
- Linux 的应用及发展前景

操作系统的基本概念

问题： 什么是操作系统？操作系统的功能是什么？典型的嵌入式操作系统有哪些？什么是进程？进程间通信有哪些方式？
重点： 操作系统的定义、功能和进程的概念。
内容： 操作系统的定义和功能，进程及进程之间的通信，典型的嵌入式操作系统简介。

要使用计算机，第一步就是要安装软件，因为没有软件的计算机是没有任何用处的。在这些软件中最基础的软件就是操作系统，如大部分人都在使用的 Windows XP，就是一个操作系统软件。如果没有安装类似于 Windows XP 的操作系统软件，也就不能安装 Microsoft Office 系列的应用软件。对于嵌入式系统来讲，如手机，实际上也是需要操作系统的。下面就将介绍操作系统的基本概念和功能。

5.1.1 操作系统的定义

任务： 从不同角度观察操作系统，掌握操作系统的定义，理解多道程序设计。

1. 从不同角度观察操作系统

实际上给操作系统下定义是困难的，至今没有一个公认的统一说法，现在从不同角度来观察一下操作系统。

(1) 从下向上看，操作系统是裸机的第一层软件，是对机器的第一次扩展，为用户提供了一台与实际硬件等价的虚拟机。

(2) 从上往下看，操作系统是计算机资源的管理者，它有序地控制着处理器、存储器以及其他 I/O 接口设备的分配，从而让系统中的多个程序能够正常地使用资源并顺利运行。

(3) 从软件分类角度看，操作系统是最基本的系统软件，它控制着计算机的所有资源并提供应用程序开发的接口。

(4) 从系统管理员角度看，操作系统能合理地组织管理计算机的工作流程，使其能为多个用户提供安全高效的计算机共享资源。

(5) 从程序员角度(即从操作系统产生的角度)看，操作系统将程序员从复杂的硬件控制中解脱出来，并为软件开发者提供了一个虚拟机，使其能更方便地进行程序设计。

(6) 从一般用户角度看，操作系统为他们提供了一个良好的交互界面，使得他们不必了解有关硬件和系统软件的细节，就能方便地使用计算机。

(7) 从硬件设计者角度看，操作系统为计算机系统功能扩展提供了支撑平台，使硬件系统与应用软件产生了相对独立性，可以在一定范围内对硬件模块进行升级和添加新硬件，而不会影响原来的应用软件。

2. 操作系统的定义

总的来讲，操作系统可以定义为：操作系统是控制和管理计算机系统内各种硬件和软件资源、合理有效地组织计算机系统的工作，为用户提供一个使用方便、可扩展的工作环境，从而起到连接计算机和用户的接口作用的软件。操作系统在计算机系统中的地位如图 5-1 所示。

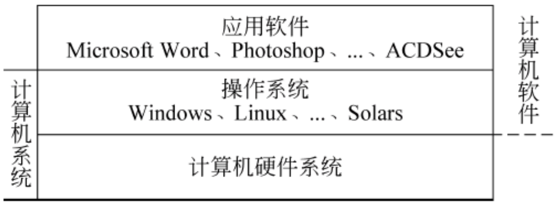


图 5-1　操作系统在计算机系统中的地位

3. 多道程序设计

现代的操作系统大多都支持多个程序同时运行,称之为多任务操作系统。例如,可以边上网聊天,边听音乐,同时还可以运行其他的应用程序,这实际上是因为操作系统应用了多道程序设计技术,使得多个程序能同时运行而不会出错。

所谓多道程序设计,是指允许多个程序同时进入内存,同时运行的技术。这样便可以充分利用系统的资源,提高系统的运行效率。图 5-2(a)所示为单道程序的运行情况。从图 5-2 中可以看出:在 t2～t3,t6～t7 时间间隔内 CPU 空闲。在引入多道程序设计技术后,由于可以把多个程序同时装入内存,当一道程序由于中断而暂停执行使得 CPU 空闲时,另一道程序可以获得 CPU 而执行,从而提高了 CPU 的利用率。图 5-2(b)所示为四道程序的运行情况。

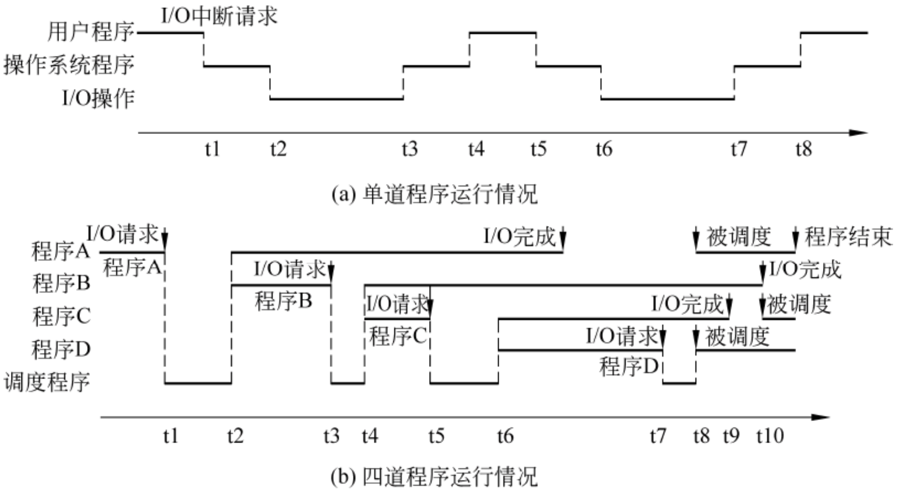


图 5-2　单道和多道程序运行情况

5.1.2　操作系统的功能

任务:了解操作系统的基本功能。

操作系统的主要功能是进行资源管理和提供方便的用户接口。计算机系统的资源可分为硬件资源和软件资源两大类。硬件资源指的是组成计算机的硬件设备,如中央处理器、主存储器、磁盘存储器、打印机、磁带存储器、显示器、键盘输入设备和鼠标等。软件资源指的是存放在计算机内的各种数据,如文件、程序库、知识库、系统软件和应用软件等。

从资源管理的观点出发,操作系统的功能可归纳为处理器管理、存储器管理、文件管理、

设备管理、用户接口。

1. 处理器管理

一个程序装入内存，必须经过处理器调度才可以运行。所谓处理器调度，就是操作系统按照一定的策略将处理器分配给等待运行的程序。在一个允许多道程序同时执行的系统里，操作系统会根据一定的策略将处理器交替地分配给系统内等待运行的程序。

实际上，由于内存中的多个程序交替使用处理器，当其不使用处理器时，会处在一个暂停的状态，当其再一次被调度时，要从暂停的地方开始运行，为了记录程序的状态，操作系统引入了进程的概念。所谓进程，就是正在运行中的程序，因此处理机调度也称为进程调度。

当一个运行中的进程遇到某个事件，例如启动外部设备而暂停执行，或发生了一个外部事件而强制其等待时，操作系统在处理完相应的事件后再重新分配处理器。

2. 存储器管理

存储器管理是指对内存资源的管理。只有被装入内存的程序才有可能去竞争处理器。因此，有效地利用主存储器可保证多道程序设计技术的实现，也就保证了处理器的使用效率。

存储器管理就是根据用户程序的要求为用户分配内存区域。当多个程序共享有限的内存资源时，操作系统就按某种分配原则，为每个程序分配内存空间，使各用户的程序和数据彼此隔离(segregate)，互不干扰(interfere)及破坏；当某个用户程序工作结束时，要及时收回它所占的内存区域，以便再装入其他程序。另外，操作系统利用虚拟内存技术，把内、外存储器结合起来，共同管理。

3. 文件管理

用户的信息一般是以文件的形式存放在磁盘中的，当需要再次使用文件时，只要提供文件的名字就可以找到该文件，这实际上是因为操作系统向用户提供了一个文件系统用来对文件进行管理，一个文件系统应该向用户提供创建文件、撤销文件、读写文件、打开和关闭文件的功能。有了文件系统后，用户可按文件名存取数据而无须知道这些数据存放在哪里。这种做法不仅便于用户使用而且还有利于用户共享公共数据。此外，由于文件建立时允许创建者规定使用权限，这就可以保证数据的安全性。

4. 设备管理

用户程序要使用外部设备，必须向操作系统进行申请，当设备满足时，由操作系统对设备进行分配，设备使用完毕，操作系统负责回收设备。设备管理的功能主要是分配和回收外部设备以及控制外部设备按用户程序的要求进行操作等。对于非存储型外部设备，如打印机、显示器等，可以直接作为一个设备分配给一个用户程序，使用完毕后要回收以便给另一个需要的用户使用。对于存储型的外部设备，如磁盘、磁带等，则用来给用户提供存储空间，用来存放文件和数据。

5. 用户接口

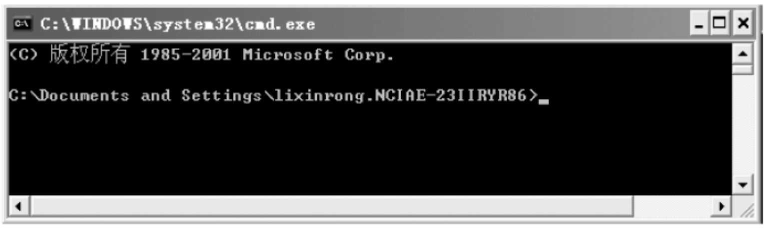
为了方便用户使用操作系统，操作系统又向用户提供了用户接口。该接口通常以命令或系统调用的形式呈现在用户面前，可分为以下3个部分。

(1) 命令接口

为了便于用户直接或间接控制自己的作业，操作系统向用户提供了命令接口。一般命令接口是由一组键盘命令组成的，例如在 Windows XP 系统中，当单击“开始”→“运行”选项

时，会打开如图 5-3(a)所示的“运行”对话框，此时只要在其中输入 cmd 命令，即可打开命令接口，如图 5-3(b)所示。在其中输入命令，即可完成相应的工作。

(a) “运行”对话框



(b) 命令接口

图 5-3　Windows XP 提供的命令接口

另外，在 Linux 操作系统中会经常使用命令接口，以便快捷地完成相应的操作。

(2) 程序接口

程序接口是操作系统提供给编程人员的接口，实际上就是操作系统提供的一组函数。例如，在编写 Windows 程序时，Windows 操作系统会向用户提供一组 API(应用程序接口)函数，以便用户使用操作系统的功能。在 Linux 操作系统中，称这种接口为系统调用，每一个系统调用都能够完成特定的功能。

(3) 图形用户接口

图形用户接口采用了图形化的操作界面，用非常容易识别的各种图标来将系统各项功能、各种应用程序和文件，直观、逼真地表示出来。例如，Windows 操作系统提供的方便快捷的图形化窗口，用户可通过鼠标、菜单和对话框来完成对应程序和文件的操作。图形用户接口元素包括窗口、图标、菜单和对话框，图形用户接口元素的基本操作包括菜单操作、窗口操作和对话框操作等。

5.1.3　操作系统的基本特征

任务：理解操作系统的基本特征。

操作系统是计算机系统的资源管理者，它协调着多个程序共同使用计算机系统的资源，由此使得操作系统具有 4 个最基本的特征：并发、共享、虚拟和异步。

1. 并发

众所周知，程序要运行，必须进行处理器调度，在只具有单核或双核 CPU 的计算机系统中，实际上，操作系统是根据一定的调度策略让多个进程在一段时间内交替使用 CPU 的。

把多件事情在同一个时间间隔内发生的情况称为并发。在计算机系统里，并发是指多个进程在一段时间内同时运行，从宏观上看所有进程都在运行，但在微观上看这些进程是处在一个“走走停停”的状态的。

另外，要把并发和并行区分开。并行是指在某个时刻同时运行，如多处理机系统，可以使多个进程并行运行，在单处理机系统中，多个进程在某一时刻只能运行一个。

2. 共享

操作系统是计算机系统资源的管理者，由于资源有限，如只有一个处理器，而多个进程都要使用，这就是共享，即系统中的资源可供内存中多个并发执行的进程共同使用。由于资源的属性不同，可将资源共享方式分为两类。

(1) 互斥共享方式。如打印机，当一个进程正在使用打印机时，其他的进程不可以使用，而处于等待状态。在一个时间段内仅供一个进程使用的资源称为临界资源。对于临界资源只能互斥访问。

(2) 同时访问方式。系统中还有另外一种资源，在一段时间内允许多个进程同时访问。典型的可供多个进程同时访问的资源是磁盘。

并发和共享是互为条件存在的。一方面，资源共享是以程序(进程)的并发执行为条件的，若系统不允许并发执行，自然不存在资源共享问题；另一方面，若系统不能对共享资源实施有效的管理，势必影响程序的并发执行，甚至根本无法并发执行。

3. 虚拟

虚拟是指一个物理实体被映射为多个逻辑意义上的实体，只是一种感觉性的存在，也就是说只是主观上的一个假象。如为了扩充内存使用的虚拟内存技术，逻辑上扩充了内存；为了共享设备，操作系统使用了虚拟设备技术；为了共享处理器，多道程序轮流使用处理器，这些技术使得用户感觉好像每道程序独自占用计算机系统的资源一样。

4. 异步

在多道程序环境下，允许多个进程并发执行，但只有在进程获得所需的资源后方能执行。在单处理机环境下，由于系统中只有一个处理机，因而每次只允许一个进程执行，其余进程只能等待。当正在执行的进程提出某种资源请求时，如打印请求，而此时打印机正在为其他某进程打印，由于打印机属于临界资源，因此正在执行的进程必须等待，且放弃处理机，直到打印机空闲，并再次把处理机分配给该进程时，该进程才能继续执行。可见由于资源等因素的限制，使进程的执行通常都不是一次完成的，而是以“走走停停”的方式运行的。

内存中的每个进程在何时可以获得处理机运行，何时又因提出某种资源请求而暂停，以及进程以怎样的速度向前推进，每道程序需要多少时间才能完成等，都是不可预知的，也就是说，进程是以人们不可预知的速度向前推进的，这种特性称为异步。

5.1.4 进程和线程的基本概念

任务： 引入进程的原因，进程的基本概念和进程的特征，线程的定义。

进程是操作系统中最基本、重要的概念，是多道程序系统出现后，为了刻画系统内部出现的动态情况，描述系统内部各道程序的活动规律引进的一个概念，所有多道程序设计操作

系统都建立在进程的基础上。

1. 进程的引入

程序在执行时，需要共享系统资源，从而导致各程序在执行过程中因争夺资源而出现相互制约的情况，程序的执行表现出间断性的特征。这些特征都是在程序的执行过程中发生的，是动态的过程，而传统的程序本身是一组指令的集合，是一个静态的概念，无法描述程序在内存中的执行情况，即无法从程序的字面上看出它何时执行，何时停顿，也无法看出它与其他执行程序的关系，因此，程序这个静态概念已不能如实反映程序并发执行过程的特征。为了深刻描述程序动态执行过程的性质，人们引入了“进程(process)”的概念。

2. 进程的概念

进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。它可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体。它不只是程序的代码，还包括当前的活动，一般是通过进程控制块(Process Control Block，PCB)来记录进程对资源的占用情况和状态的。进程控制块实际上是在程序装入内存时由操作系统创建的一个数据结构。如在 Linux 系统中，一个进程的控制块用一个称为 task_struct 的结构体来描述，下面是部分代码。

```
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void * stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;
    int lock_depth; /* BKL lock depth */
#ifdef CONFIG_SMP
#ifdef __ARCH_WANT_UNLOCKED_CTXSW
int oncpu;
#endif
#endif
    int prio, static_prio, normal_prio;
    struct list_head run_list;
    const struct sched_class * sched_class;
    struct sched_entity se;
#ifdef CONFIG_PREEMPT_NOTIFIERS/* list of struct preempt_notifier: */
struct hlist_head preempt_notifiers;
#endif
    unsigned short ioprio;
    /*
     * fpu_counter contains the number of consecutive context switches
     * that the FPU is used. If this is over a threshold, the lazy fpu
     * saving becomes unlazy to save the trap. This is an unsigned char
     * so that after 256 times the counter wraps and the behavior turns
     * lazy again; this to deal with bursty apps that only use FPU for
     * a short time
     */
    unsigned char fpu_counter;
    s8 oomkilladj; /* OOM kill score adjustment (bit shift). */
#ifdef CONFIG_BLK_DEV_IO_TRACE
unsigned int btrace_seq;
```

```
#endif
…
```

在进程控制块中包含了进程的运行状态、进程的通信状况、进程的标号、内存的占用情况等。每一个 PCB 都是这样的，只有采用这些结构才能满足一个进程的所有要求。

例如，课程和上课的关系，课程是一个静态的概念，当把上课的时间、教师、教室分配好后就可以上课了。上课是一个动态的概念，只有分配了相应的资源后才可以正常运行。课程表就相当于进程控制块，它记录着上课的时间和地点等信息。

3. 进程的特征

进程是程序在处理器上的一次执行过程，具有一定的生命周期，并且多个进程可以并发执行，因此进程具有如下特征。

(1) 动态性

进程的实质是程序的一次执行过程，进程是动态产生、动态消亡的。

(2) 并发性

任何进程都可以同其他进程一起并发执行。

(3) 独立性

进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位。

(4) 异步性

由于进程间相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。

(5) 结构特征

进程由程序、数据和进程控制块 3 部分组成。

4. 线程的概念

在现代操作系统中，允许一个进程中同时运行多个线程，这样的程序称为多线程程序。所有的程序都有一个主线程(main thread)，主线程是进程的控制流或执行线程，如图 5-4 所示。

线程是进程的一条执行路径，它包含独立的堆栈和 CPU 寄存器状态，每个线程共享其所附属的进程的所有资源。线程和进程的关系是：线程是属于进程的，线程运行在进程空间内，同一进程所产生的线程共享同一物理内存空间，当进程退出时该进程所产生的线程都会被强制退出并清除。

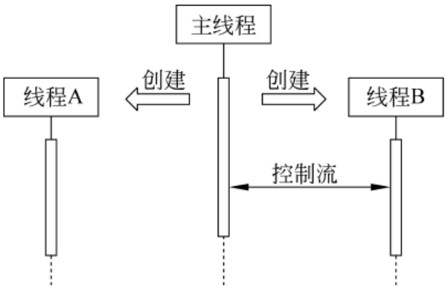


图 5-4 多线程进程的控制流

5.1.5 进程的同步与互斥

任务：理解进程之间的关系，掌握互斥与同步的定义。

在以进程为基础的管理机制下，任务被分解成了多个进程，而一个进程可能只完成某个任务中的一部分，多个进程共同执行才能顺利完成一项任务。这就产生了以下问题：进程在执行中有哪些约束和限制，进程在执行中需要采用什么方式彼此交互信息，以保证共享信息的正确使用和进程的并发执行。从另一个角度来看进程间的相互关系，可以将进程之间

的关系分为同步与互斥两种。

在操作系统中,一般是通过进程的同步与互斥机制来完成进程间的相互合作与约束的。

1. 进程互斥

进程互斥是指多个进程共享某个临界资源时,为了保证共享资源能够被正确地使用,当一个进程正在使用这个资源时,不允许其他进程使用,这种情况称为进程互斥。图 5-5 所示是一个进程互斥的例子。对于临界资源文件或打印机,当有多个进程请求时只可能有一个进程的请求被接受(如进程 k),其他进程的请求都会被拒绝。

2. 进程同步

进程同步是指系统中的多个进程之间存在某种时序关系,需要相互协作与制约,才能共同完成一项任务。例如,有一对计算进程和打印进程,它们共享着一个缓冲区 Buf。计算进程不断向缓冲区中写数据,但要求写入数据之前缓冲区必须是空的;打印进程不断从缓冲区中取出数据进行打印,而且每当完成一次打印后就清空缓冲区。计算进程和打印进程可以分别独立地运行,两者之间有相互制约和相互协作的关系,两个进程之间的同步关系如图 5-6 所示。

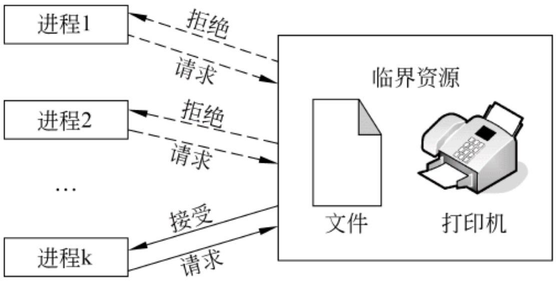


图 5-5 进程以互斥方式访问临界资源示意图

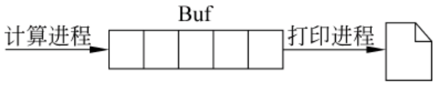


图 5-6 计算进程与打印进程的同步关系

中断和中断处理

问题: 什么是中断?操作系统为什么需要中断?操作系统怎样处理中断,怎样响应中断?
重点: 中断的定义,中断处理与中断返回,中断响应和处理过程。
内容: 中断、中断处理和中断响应。

5.2.1 中断

任务: 理解什么是异步事件,掌握与中断相关的基本概念,包括中断、中断源、中断向量和中断优先级。

1. 异步事件

所谓异步事件,就是与当前正在做的事情在逻辑上无关的事件。例如,老师正在上课时,突然发生了地震,此时上课被迫停止,地震就是一个异步事件,它与上课没有必然联系。在计算机系统中,异步事件通常是指由外部设备产生的事件。例如,程序正在执行时按了主机上的 Reset 键,使当前正在运行的程序停止运行。

2. 中断

在计算机技术中,由于某种异步事件的发生而使程序执行流程发生转移的现象叫做中断。

产生异步事件的原因叫做中断源。中断源在发生异步事件时都会向处理器发出一个通知信号,该信号叫做中断请求信号。处理器对这个中断请求信号可以响应,也可以不响应。如果响应了该信号,则处理器的执行流程将会保存当前程序的断点,然后跳转到另外一个叫做中断服务程序的程序上去执行,这个过程叫做中断响应。

3. 中断向量和中断优先级

中断向量即中断源的识别标志,可用来存放中断服务程序的入口地址或跳转到中断服务程序的入口地址。为了处理上的方便,通常为每种设备配以相应的中断处理程序,并把所有中断处理程序的入口地址保存在一个表中,这个表叫做中断向量表。

在中断向量表中为每一个设备的中断请求规定一个中断号,当有中断发生时,根据中断号找到相应的中断程序入口地址,这样便可以转入中断处理程序执行。

在实际情况下,经常会有多个中断信号源,每个中断源对服务要求的紧急程度并不相同。例如,在一个飞机控制系统中打开飞机起落架的紧急程度就比打开空调的紧急程度要高,为此,系统为它们分别规定不同的优先级,当有多个异步事件同时发生时,优先级高的事件首先得到处理。

5.2.2 中断处理与中断返回

任务:掌握中断响应和中断的过程。

中断处理是指响应中断后,CPU 执行中断服务程序对中断进行处理的过程,这个处理过程由程序(软件)实现,如图 5-7 中的虚线框部分所示。

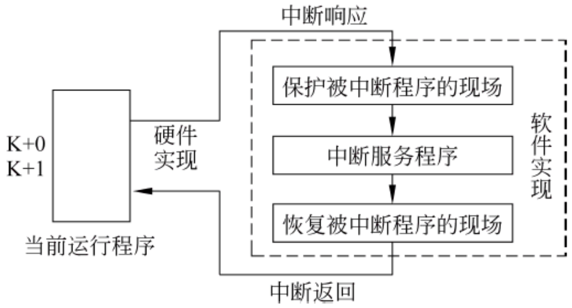


图 5-7 中断响应和处理过程

中断服务程序的处理过程一般包括以下 3 个步骤。

(1) 保护被中断程序的现场。

(2) 进行中断处理或提供中断服务。

(3) 恢复被中断程序的现场,中断返回。

通常,"被中断程序的现场"包括在中断服务程序中用到的寄存器中,它们都需要在执行中断服务程序之前进行保护,以免由于中断服务程序对它们的修改而破坏被中断程序存放在其中的数据,从而导致中断返回后被中断的程序不能正常地执行。

中断服务完成后可以使用系统提供的中断返回指令返回被中断的程序的断点(K+0)的下一条指令(返回点 K+1)继续执行。

所以,中断过程是由硬件和软件配合完成的,硬件实现中断响应,软件实现中断处理。

单内核与微内核

问题: 操作系统的内核是什么? Linux 操作系统的内核分为哪几部分? 目前常见的内核类型有哪些?

重点: Linux 操作系统内核的组成,内核的分类:单内核(宏内核)和微内核及二者的比较。

内容: 操作系统的内核是什么,Linux 内核的组成,常见的内核分类。

5.3.1　内核

任务: 掌握内核的组成,了解 Linux 内核结构。

内核是操作系统最基本的部分,它由操作系统中用于管理存储器、文件、外围设备和系统资源的部分组成,指的是一个提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件。很多应用程序对计算机硬件的安全访问要通过内核实现,一个程序在什么时候对某部分硬件操作多长时间也要由内核决定。直接对硬件操作是非常复杂的,所以内核通常提供一种硬件抽象的方法来完成这些操作。硬件抽象隐藏了复杂性,为应用软件和硬件提供了一套简洁、统一的接口,使程序设计更为简单。如图 5-8 所示为 Linux 内核结构框图。

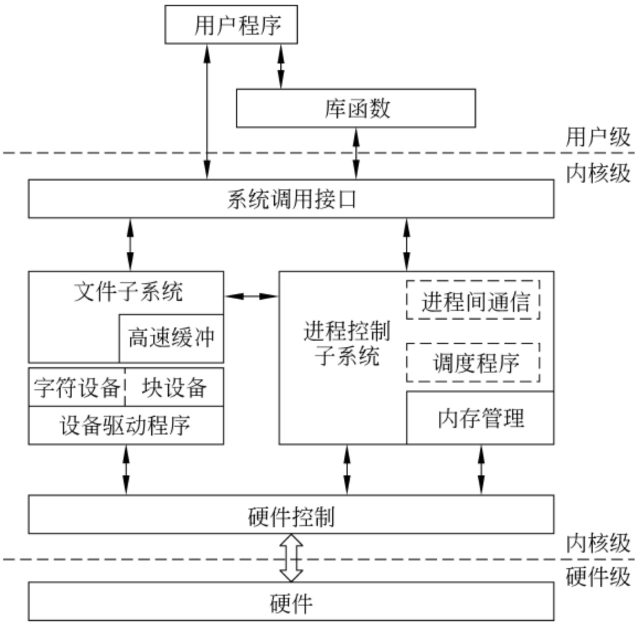


图 5-8　Linux 内核结构框图

Linux 内核中设置了一组用于实现各种系统功能的子程序,称为系统调用。用户可以

通过系统调用命令在自己的应用程序中调用它们。从某种角度来看,系统调用和普通的函数调用非常相似。区别仅在于,系统调用由操作系统核心提供,运行于核心态;而普通的函数调用由函数库或用户自己提供,运行于用户态。

随 Linux 核心还提供了一些 C 语言函数库,这些库对系统调用进行了一些包装和扩展,这些库函数与系统调用的关系非常紧密,实际上,人们熟悉的很多 C 语言标准函数在 Linux 平台上的实现都是靠系统调用完成的。

5.3.2 单内核操作系统与微内核操作系统

任务:单内核操作系统和微内核操作系统的特点。

内核是操作系统最基础的构件,因而,内核结构往往对操作系统的外部特性以及应用领域有着根本的影响。内核的结构往往可分为单内核(monolithic kernel)、微内核(microkernel)、超微内核(nanokernel),以及外内核(exokernel)等。超微内核与外内核等其他结构是于 20 世纪末在理论界发展起来的,大部分时候只在实验室使用;而自 20 世纪 80 年代起,大部分理论研究都集中在以微内核为首的"新兴"结构之上;同时,在应用领域之中,以单内核结构为基础的操作系统却一直占据着主导地位。图 5-9 是单内核操作系统和微内核操作系统结构比较示意图。

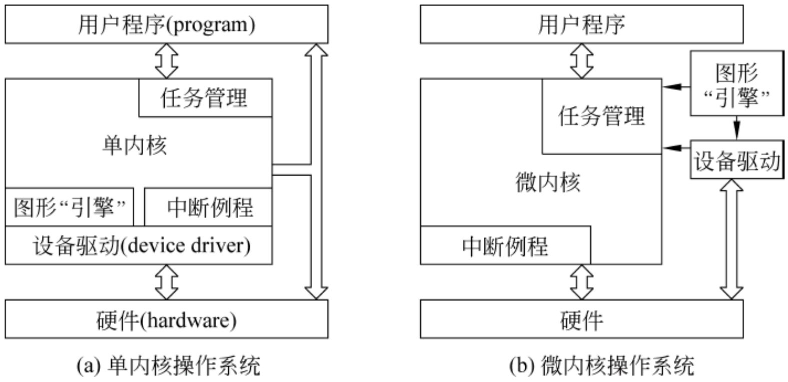


图 5-9 单内核操作系统和微内核操作系统结构比较示意图

1. 单内核操作系统

单内核操作系统使用传统的模块化设计技术,根据实现的功能将系统划分为不同的模块,模块之间的通信方法是直接的函数调用。单内核结构在内核中集中了全部或大部分的系统服务或系统功能,如图 5-9(a)所示。单内核的好处是极大地减少了在系统服务间的上下文切换(context switch)和系统服务间的消息引用(message involved)上花费的时间,从而在理论上获得比微内核结构更快的应用处理速度。基于单内核的操作系统通常有着较长的历史渊源。例如,绝大部分 UNIX 的家族史都可追溯至 20 世纪 60 年代。该类操作系统多数有着相对古老的设计和实现(如某些 UNIX 中存在着大量 20 世纪七八十年代的代码)。当今,占主导地位的操作系统如 Linux、大部分的 UNIX 等都属于单内核的操作系统结构。

2. 微内核操作系统

在微内核操作系统中内核被设计得尽可能简单,微内核将大部分的系统服务如网络服

务、文件系统等作为代理进程或服务进程运行在用户态或用户空间;内核只提供一些非常基本的系统服务,如内存分配、进程调度和消息处理等,如图 5-9(b)所示。因为运行在内核空间的代码被大大减少了,也大大提高了内核的稳定性。然而,因为一个程序的实现常常需要多次系统调用,于是在这种结构的操作系统中,一个功能调用要经历多次进程的上下文切换,并且要引发多次消息转换。这样,从整体上来说,就使得微内核结构的操作系统在理论上要比单内核的操作系统慢(当然,这并不是说一个设计巧妙的微内核系统一定会比一个设计普通的单内核系统慢)。

3. 二者的比较

微内核操作系统特有的架构的很多特点正好匹配了嵌入式平台对操作系统的需求,非常适合于嵌入式环境的应用。

首先是可靠性,按照单内核操作系统的设计,内核包括所有的操作系统服务,其中任何一个服务出错,都会造成整个系统的崩溃。微内核操作系统的设计思想是在内核中留尽量少的东西,只保留实现操作系统服务最基本的机制,而把具体服务的实现放到用户态的服务应用程序中去,这就大大降低了内核崩溃的几率。特别是目前操作系统的许多错误都是因为不规范并且没有经过严格测试的驱动程序造成的。

但微内核架构也不是完美的,它有一个很大的缺点,那就是性能问题。对于单内核操作系统,调用系统服务的方式是通过系统调用,需要的仅仅是用户态和内核态的两次转换,每个进程都同时有用户栈和内核栈,可以存放执行过程中的信息。而对于微内核操作系统,需要通过发送 IPC 消息给服务应用程序调用系统服务,服务应用程序通过系统调用完成服务请求后再通过另一个 IPC 消息把结果返回给调用者,这涉及进程的上下文切换,并且由于没有内核栈这样简单的机制,传送消息需要额外的复制开销,因此性能成为微内核架构操作系统的一个很大的问题。

事实上,在嵌入式系统中,选择单内核操作系统还是微内核操作系统,目前很难下定论,在选择时,应根据综合情况进行判定。例如,从开源的角度来看,选择 Linux 单内核操作系统节省资金;从安全性上来讲,选择微内核的操作系统可能更加可靠。

操作系统的类型

问题: 操作系统的类型有哪些? 各自有哪些优缺点?
重点: 操作系统的分类及其使用范围和各自的优缺点。
内容: 单用户操作系统、批处理操作系统、分时操作系统、实时操作系统和嵌入式操作系统及其发展,常用的典型操作系统、优缺点及使用范围。

操作系统是计算机系统软件的核心,有多种分类方法。按照操作系统的发展进程,可以将操作系统分为单用户操作系统、批处理操作系统、分时操作系统、实时操作系统和嵌入式操作系统等。

5.4.1 单用户操作系统

任务: 了解单用户操作系统的特征及其特点。

单用户操作系统是随着微型计算机的发展而产生的，用来对一台计算机的硬件和软件资源进行管理，通常分为单用户单任务和单用户多任务两种类型。

单用户单任务操作系统的主要特征是：在一个计算机系统内，一次只能运行一个用户程序，此用户独占计算机系统的全部硬件和软件资源。常用的单用户单任务操作系统有MS-DOS、PC-DOS等。

单用户多任务操作系统也是为单个用户服务的，但它允许用户一次提交多项任务。例如，用户可以在运行程序的同时开始另一文档的编辑工作，边听音乐边打字也是一个典型的例子。常用的单用户多任务操作系统有OS/2、Windows 95/98等，这类操作系统通常用在微型计算机系统中。

单用户操作系统一次只能支持一个用户进程的运行，相对于多用户操作系统它可以支持多个用户同时登录，允许运行多个用户的进程。例如，Windows XP本身就是一个多用户操作系统，不管是在本地还是远程都允许多个用户同时处于登录状态。

特点：单用户，不要求高利用率，具有良好的交互性。

5.4.2 批处理操作系统

任务：了解批处理操作系统的基本思想及其特点。

批处理操作系统是早期配置在大型机上的一种操作系统，其特点是用户脱机使用计算机、作业成批处理和多道程序运行。

批处理操作系统要求用户事先把上机的作业准备好，其中包括程序、数据以及作业说明书，然后直接交给系统操作员，并按指定的时间收取运行结果，用户不直接与计算机交互。系统操作员不会立即进行输入作业，而是要等到一定时间或作业达到一定数量之后才进行成批输入。系统操作员对用户提交的作业分批进行处理，每批中的作业由操作系统控制执行。

批处理系统可以分为简单批处理系统和多道批处理系统。多道批处理系统是多道程序设计技术与批处理系统的结合，基本思想是每次把一批经过合理搭配的作业通过输入设备提交给操作系统，并暂时存入外存，等待运行。当系统需要调入新的作业时，根据当时的运行情况和用户要求，按某种调度原则，从外存中挑选一个或几个作业装入内存运行。用户用控制命令描述对作业每一步运行的具体安排，并将这些控制连同程序和数据一起作为一个作业交给操作系统，因此，在系统运行过程中不允许用户与其作业有交互作用，即用户不能直接干预自己作业的运行，直到作业运行完毕。

多道批处理系统一般用于较大的计算机系统，要求较高的利用率和吞吐量，例如，OS/360 MTV是一个典型的多道批处理操作系统。

特点：不允许用户介入，没有交互性，单道批处理系统的资源利用率低于多道批处理系统的利用率。

5.4.3 分时操作系统

任务：了解分时操作系统的基本思想和分时系统的主要目的。

分时操作系统的基本思想是让多个用户通过终端同时使用系统，而操作系统则在内部

按一定策略调度和处理用户程序。即允许多个用户共享同一台计算机的资源，在一台计算机上连接几台甚至几十台终端机，终端机可以没有 CPU 与内存，只有键盘和显示器，每个用户都通过各自的终端机使用这台计算机的资源，计算机系统按固定的时间片轮流为各个终端服务。由于计算机的处理速度很高，人的反应要比机器慢得多，所以用户感觉不到等待时间，多个用户都会觉得这台计算机是专为自己服务的。分时系统的控制示意图如图 5-10 所示。

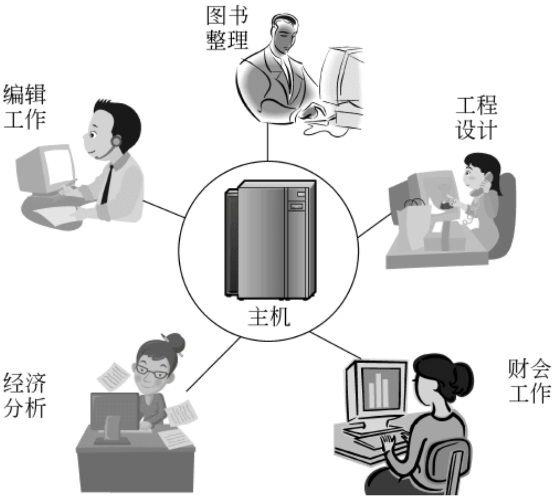


图 5-10　分时系统的控制

分时操作系统的主要目的是对联机用户的服务响应，具有同时性、独立性、及时性和交互性等特点。在分时操作系统中，分时是指若干道程序分享 CPU 运行，通过设立一个单位时间片来实现。也就是说 CPU 按时间片轮流执行各个作业，一个时间片通常是几十毫秒。

分时系统是在多道批处理系统的基础上发展起来的，在分时系统中，用户通过计算机交互会话来联机控制作业运行，一个分时系统可以带几十个甚至上百个终端，每个用户都可以在自己的终端上操作或控制作业的完成，从宏观上看，多用户同时工作，共享系统资源；从微观上看，各进程按时间片轮流运行，提高了系统资源利用率。

CTSS 是最早的分时操作系统，在现今流行的操作系统中，Linux、Windows、OS/2 以及 UNIX 都是分时系统。

特点：多用户、交互性好。

5.4.4　实时操作系统

任务：掌握实时操作系统的定义，能够列举出实时系统的实例。

实时操作系统是随着工业过程控制和对信息进行实时处理的需要而产生的。“实时”是“立即”的意思，是指对随机发生的外部事件(指来自于计算机系统相连接的设备所提出的服务要求)做出及时的响应并对其进行处理。

能使计算机系统接收到外部信号后及时进行处理，并且在严格的规定时间内处理结束，再给出反馈信号的操作系统称为“实时操作系统”，简称“实时系统”。

图 5-11 所示是一个实时系统的示例。在图中，有一艘舰船，在舰船的头部装有用来发

现礁石的声纳，以免发生触礁事故。现用一个计算机系统接收并处理声纳信号，计算机的计算结果用来控制舵机的动作。假设声纳在舰船前面发现礁石而舰船的舵机不采取任何规避动作，则舰船将在 10min 后与礁石相撞。现在假设，为了防止触礁，舵机完成合理的规避动作需要 8min，那么，从声纳发现礁石起到舵机开始动作，留给计算机系统用来计算和控制舵机做出合理动作所需结果的时间不能超过 2min，并且应保证计算结果正确无误；否则，结果将是灾难性的。

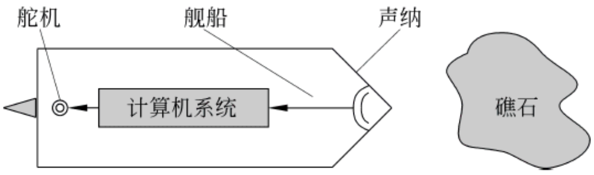


图 5-11 实时系统的示例

实时操作系统主要是为联机实时任务服务的，以在允许的时间范围内做出响应为主要特征，要求计算机对外来的信息能以足够快的速度进行处理，并在被控对象允许的时间范围内做出响应，其响应时间在秒级、毫秒级或微秒级甚至更小，通常用在工业过程控制和信息实时处理方面。工业控制主要包括数控机床、电力生产系统、导弹的制导系统等方面的自动控制；信息实时处理主要包括民航中的飞机订票系统、铁路订票系统、电子银行系统等。

特点：响应速度快，具有高度的可靠性和安全性。

当今流行的嵌入式操作系统简介

问题： 当今流行的嵌入式操作系统有哪些？在嵌入式系统中为什么要使用操作系统？怎样选择嵌入式操作系统？

重点： 常见的嵌入式操作系统及其优缺点。

内容： 嵌入式操作系统的发展，使用嵌入式操作系统的必要性，嵌入式操作系统的选型，常见的开源和商业嵌入式操作系统简介。

5.5.1 嵌入式操作系统的发展

任务： 了解嵌入式操作系统发展的 4 个阶段。

其实，嵌入式系统并不是一个新生的事物，从 20 世纪 80 年代起，国际上就有一些 IT 组织、公司开始进行商用嵌入式系统和专用操作系统的研发，这期间涌现了一些著名的嵌入式操作系统：Windows CE、VxWorks、pSOS、QNX、Palm OS、OS-9、LynxOS。

目前，有很多商业性嵌入式系统都在努力地为自己争取嵌入式市场的份额。但是，这些专用操作系统均属于商业化产品，价格昂贵；而且，由于它们各自的源代码不公开，使得每个系统上的应用软件与其他系统都无法兼容。并且，由于这种封闭性还导致了商业嵌入式系统在对各种设备的支持方面存在很大的问题，使得对它们的软件移植变得很困难。在嵌入式这个 IT 产业的新的关键领域，Linux 操作系统适时地出现了，由于 Linux 自身诸多的优

势,吸引了许多开发商的目光,成为嵌入式操作系统的新宠。它的出现无疑为发展嵌入式操作系统事业提供了一个极有吸引力的机会。

嵌入式操作系统伴随着嵌入式系统的发展经历了 4 个阶段。

第一阶段：无操作系统的嵌入算法阶段,以单芯片为核心的可编程控制器形式的系统,具有与监测、伺服、指示设备相配合的功能。应用于一些专业性极强的工业控制系统中,使用汇编语言编程对系统进行直接控制,运行结束后清除内存。系统结构和功能都相对单一,处理效率较低,存储容量较小,几乎没有用户接口。

第二阶段：以嵌入式 CPU 为基础、简单操作系统为核心的嵌入式系统。CPU 种类繁多,通用性比较差;系统开销小,效率高;一般配备系统仿真器,操作系统具有一定的兼容性和扩展性;应用软件较专业,用户界面不够友好;系统主要用来控制系统负载以及监控应用程序运行。

第三阶段：通用的嵌入式实时操作系统阶段,以嵌入式操作系统为核心的嵌入式系统。能运行于各种类型的微处理器上,兼容性好;内核精小,效率高,具有高度的模块化和扩展性;具备文件和目录管理、设备支持、多任务、网络支持、图形窗口以及用户界面等功能;具有大量的应用程序接口(API);嵌入式应用软件丰富。

第四阶段：以 Internet 为标志的嵌入式系统。这是一个正在迅速发展的阶段。目前大多数嵌入式系统还孤立于 Internet 之外,但随着 Internet 的发展以及 Internet 技术与信息家电、工业控制技术等的结合日益密切,嵌入式设备与 Internet 的结合代表着嵌入式技术的真正未来。

5.5.2　使用嵌入式操作系统的必要性

任务：掌握在嵌入式操作系统中使用操作系统的必要性。

嵌入式实时操作系统在目前的嵌入式应用中用得越来越广泛,尤其在功能复杂、系统庞大的应用中显得愈来愈重要。

(1) 嵌入式实时操作系统提高了系统的可靠性

在控制系统中,出于安全方面的考虑,要求系统起码不能崩溃,而且还要有自愈能力。不仅要求在硬件设计方面提高系统的可靠性和抗干扰性,而且也应在软件设计方面提高系统的抗干扰性,尽可能地减少安全漏洞和不可靠的隐患。例如,有些系统软件在运行时,可能会遇到强干扰,从而产生异常或出错,甚至造成系统崩溃。而嵌入式操作系统则可以通过系统监控进程对其进行修复,从而提高嵌入式操作系统的可靠性。

(2) 提高了开发效率缩短了开发周期

在嵌入式实时操作系统环境下,开发一个复杂的应用程序,通常可以将整个程序分解为多个任务模块。每个任务模块的调试、修改几乎不影响其他模块。商业软件一般都提供了良好的多任务调试环境。

(3) 嵌入式实时操作系统充分发挥了 32 位 CPU 的多任务潜力

32 位 CPU 比 8 位、16 位 CPU 快,另外它本来是为运行多用户、多任务操作系统而设计的,特别适于运行多任务实时系统。32 位 CPU 采用利于提高系统可靠性和稳定性的设计,使其更容易做到不崩溃。例如,CPU 运行状态分为系统态和用户态。将系统堆栈和用户堆

栈分开,以及实时地给出 CPU 的运行状态等,允许用户在系统设计中从硬件和软件两方面对实时内核的运行实施保护。从某种意义上说,没有操作系统的计算机(裸机)是没有用的。在嵌入式应用中,只有把 CPU 嵌入到系统中,同时把操作系统嵌入进去,才是真正的嵌入式系统。

5.5.3 嵌入式操作系统选型

任务:了解选择操作系统应遵循的原则。

当设计信息电器、数字医疗设备等嵌入式产品时,嵌入式操作系统的选择至关重要。一般而言,在选择嵌入式操作系统时,可以遵循以下原则。

(1) 市场进入时间

嵌入式产品能否迅速进入市场与选择操作系统有关,例如,Windows 的程序员可能是人力资源最丰富的,使得使用 Windows CE 开发的嵌入式产品能够很快进入市场。但某些高效的操作系统可能会由于编程人员缺乏或这方面的技术积累不够,而影响开发进度,使得产品进入市场时间后移。

(2) 可移植性

当进行嵌入式软件开发时,可移植性是要重点考虑的问题。具有良好的软件移植性,可以在不同平台、不同系统上运行,跟操作系统无关。软件的通用性和软件的性能通常是矛盾的。即通用性是以损失某些特定情况下的优化性能为代价的。很难设想开发一个嵌入式浏览器而仅能在某一特定环境下应用。反过来说,当产品与平台和操作系统紧密结合时,往往产品的特色就蕴含其中。

(3) 可利用资源

产品开发不同于学术课题研究,它是以快速地推出满足用户需求的低成本、高质量产品为目的的。集中精力研发出产品的特色,其他功能尽量由操作系统附加或采用第三方产品,因此操作系统的可利用资源对于选型是一个重要参考条件。Linux 和 Windows CE 都有大量的资源可以利用,这是它们被看好的重要原因。其他有些实时操作系统由于比较封闭,开发时可以利用的资源比较少,因此多数功能需要自己独立开发,从而影响开发进度。近来的市场需求显示,越来越多的嵌入式系统,均要求提供全功能的 Web 浏览器,而这要求有一个高性能、高度可靠的 GUI 的支持。

(4) 系统定制能力

信息产品不同于传统 Intel 结构的 PC,用户的需求是千差万别的,硬件平台也都不一样,所以对系统的定制能力提出了要求。要分析产品是否对系统底层有改动的需求,这种改动是否伴随着产品特色,Linux 由于其源代码开放,在定制能力方面具有优势。随着 Windows CE 源码的开放,以及微软在嵌入式领域力度的加强,其定制能力还会有所提升。

(5) 成本

成本是所有产品不得不考虑的问题。操作系统的选择会对成本产生一定的影响。Linux 免费,Windows CE 等商业系统需要支付许可证使用费,但这不会对成本产生什么影响。成本是需要在综合权衡以后进行考虑的——选择某一系统可能会对其他一系列的因素产生影响,如对硬件设备的选型、人员投入,以及公司管理和与其他合作伙伴共同开发时的

沟通等许多方面的影响。

(6) 中文内核支持

国内产品需要提供对中文的支持。由于操作系统多数采用西文方式,是否支持双字节编码方式,是否遵循 GBK、GB 18030 等各种国家标准,是否支持中文输入与处理,是否提供第三方中文输入接口是针对国内用户的嵌入式产品必须考虑的重要因素。

上面提到用 Windows CE+x86 出产品是减法,这实际上就是所谓的 PC 家电化;另外一种做法是加法,利用家电行业的硬件解决方案(绝大部分是非 x86 的)加以改进,加上嵌入式操作系统,再加上应用软件,这就是所谓的家电 PC 化的做法,这种加法的优势是成本低、特色突出,缺点是产品研发周期长、难度大(需要深入了解硬件和操作系统),如果采用这种做法,Linux 是一个好的选择,因为能够深入到系统底层。

5.5.4 常见的开源嵌入式操作系统简介

任务:了解常见的开源嵌入式操作系统的特点。

1. uC/OS-II

uC/OS-II 是一种公开源代码、结构小巧、实时内核的实时操作系统,自从 1992 年发布以来,在世界各地都得到了广泛的应用。uC/OS-II 是一种基于优先级的可抢先的硬实时内核,其内核提供任务调度与管理、时间管理、任务间同步与通信、内存管理和中断服务等功能。它适用于小型控制系统,具有执行效率高、占有空间小、实时性能优良和扩展性强等特点,最小内核可以编译至 2KB 左右。目前已经被移植到 40 多种不同架构的 CPU 上,运行在从 8 位到 64 位的各种系统之上。尤其值得一提的是,该系统自从 2.51 版本之后,就通过了美国 FAA 认证,可以运行在诸如航天器等对安全要求极为苛刻的系统之上。鉴于 uC/OS-II 可以免费获得代码,对于嵌入式 RTOS 而言,选择 uC/OS-II 无疑是最经济的选择。

2. RTLinux

RTLinux 是源代码开放的具有硬实时特性的多任务操作系统,它是通过底层对 Linux 实施改造的产物。通过在 Linux 内核与硬件中断之间增加一个精巧的可抢先的实时内核,将标准的 Linux 内核作为实时内核的一个进程与用户进程一起调度,标准的 Linux 内核的优先级最低,可以被实时进程抢断。正常的 Linux 进程仍可以在 Linux 内核上运行,这样既可以使用标准分时操作系统即 Linux 的各种服务,又能提供低延时的实时环境。

RTLinux 已成功地应用于从航天飞机的空间数据采集、科学仪器测控到电影特技图像处理等广泛的实时环境下。

3. ARM-Linux

ARM-Linux 就是在 ARM 芯片上应用的嵌入式实时操作系统。近年来,由于嵌入式微处理器核 ARM 在世界范围内都有广泛的应用,Linux 也被移植到了 ARM 体系的芯片上了。尤其在我国,有关 ARM-Linux 的应用和移植的资料非常多,为学习嵌入式系统开发提供了有力的支持。

4. uCLinux

uCLinux 是 Lineo 公司的主打产品,同时也是开放源码的嵌入式 Linux 的典范之作。uCLinux 主要是针对目标处理器没有存储管理单元 MMU (Memory Management Unit)的

嵌入式系统而设计的。它已经被成功地移植到了很多平台上。由于没有 MMU,其多任务的实现需要一定技巧。uCLinux 是一种优秀的嵌入式 Linux 版本,是 micro-Conrol-Linux 的缩写。它秉承了标准 Linux 的优良特性,经过各方面的小型化改造,形成了一个高度优化的、代码紧凑的嵌入式 Linux。虽然它的体积很小,却仍然保留了 Linux 的大多数优点:稳定、良好的移植性;优秀的网络功能;对各种文件系统完备的支持和标准丰富的 API。它专为嵌入式系统做了许多小型化的工作,目前已支持多款 CPU。其编译后的目标文件可控制在几百 KB 数量级,并已经被成功地移植到很多平台上。

5.5.5 常见的商业嵌入式操作系统简介

任务:了解常见的商业嵌入式操作系统的特点。

1. Windows CE

Windows CE 是微软开发的一个开放的、可升级的 32 位嵌入式操作系统,是针对掌上型电脑类电子设备开发的。Windows CE 是所有源代码全部由微软自行开发的新型嵌入式操作系统,它不仅继承了传统的 Windows 图形界面,并且在 Windows CE 平台上可以使用 Windows 95/98 上的编程工具(如 Visual Basic、Visual C++),使用同样的函数和同样的界面风格,使绝大多数的应用软件只需简单地修改和移植就可以在 Windows CE 平台上继续使用。

Windows CE 模块化及可伸缩性、实时性好,通信能力强大,支持多种 CPU 是其基本的设计目标。它的设计可以满足多种设备的需要,这些设备包括工业控制器、通信集线器以及销售终端之类的企业设备,还有像照相机、电话和家用娱乐器材之类的消费产品。一个典型的基于 Windows CE 的嵌入系统通常为某个特定用途而设计,并在不联机的情况下工作。它要求所使用的操作系统体积较小,内建有对中断的响应功能。

Windows CE 除具有灵活的电源管理、对象存储技术、良好的通信能力、线程响应能力等特点之外,Windows CE 还提供了相当丰富的编程组件,如在掌上型电脑中,Windows CE 就包含了如远程拨号访问、世界时钟、计算器、输入法、中文字库、袖珍浏览器、电子邮件、Pocket Office 等多种组件。这些特点使得 Windows CE 使用起来更方便,进一步提高了其市场占有率。

2. VxWorks

VxWorks 操作系统是美国 WindRiver 公司于 1983 年设计开发的一种嵌入式实时操作系统(RTOS),是 Tornado 嵌入式开发环境的关键组成部分。良好的持续发展能力、高性能的内核以及友好的用户开发环境,在嵌入式实时操作系统领域逐渐占据一席之地。

VxWorks 具有可裁剪微内核结构;实现了高效的任务管理、灵活的任务间通信、微秒级的中断处理;支持 POSIX 1003.1b 实时扩展标准;支持多种物理介质及标准的、完整的 TCP/IP 网络协议等。然而其价格昂贵。由于操作系统本身以及开发环境都是专有的,价格一般都比较高,通常需花费 10 万元人民币以上才能建起一个可用的开发环境,对于每一个应用一般还要另外收取版税。一般不提供源代码,只提供二进制代码。由于它们都是专用操作系统,需要专门的技术人员掌握开发和维护技术,所以软件的开发和维护成本都非常高。支持的硬件数量有限。

VxWorks 是目前嵌入式系统领域中使用最广泛、市场占有率最高的系统。它支持多种处理器，如 x86、i960、Sun Sparc、Motorola MC68xxx、MIPS RX000、PowerPC 等。大多数的 VxWorks API 是专有的。采用 GNU 的编译器和调试器。

3. MontaVista Linux

MontaVista 专门提供实时嵌入式 Linux 操作系统和嵌入式开发工具，虽然它进入中国的时间很短，但它的大名早就为国人所熟知，而且通常会将它的 MontaVista Linux（即原来的 Hard Har Linux）与 VxWorks 和 Windows CE 相提并论。

不同的是 MontaVista Linux 是基于 Linux 内核开发的嵌入式操作系统。相对于一些专用的嵌入式操作系统，MontaVista Linux 有自己的优势：MontaVista Linux 不需要用户支付版税，而且 MontaVista Linux 提供的所有开发工具和附加应用包都是开放源码的；MontaVista Linux 基于 Linux 内核，而 Linux 是从 UNIX 发展而来的，所以它很稳定；MontaVista Linux 能够支持广泛的 CPU 芯片系列，支持多种目标板结构，并提供强大的网络协议支持，而且拥有丰富的驱动程序和 API。

正是因为 MontaVista Linux 是基于 Linux 开发的，MontaVista Linux 的所有源代码都是对外开放的，所以 MontaVista Linux 提供产品的方式与常规产品不同。客户可以不花一分钱得到 MontaVista Linux 的所有源代码，并可以对其进行任何修改。但如何使用这些代码，对于要在其基础上进行应用开发的客户来说，通常会显得过于复杂，尤其是嵌入式应用的复杂性较高，两个嵌入式系统对操作系统的剪裁可能会截然不同，所以更为专业的 MontaVista Linux 技术人员可以为产品提供更准确的剪裁和整合。MontaVista Linux 通常是通过以下方式向客户提供需要的产品和服务的：客户根据自己的实际应用情况向 MontaVista 订阅产品，从而获得不同级别和年限的订阅，然后 MontaVista 根据客户的实际需求对自己的产品进行剪裁和集成，并在经过严格测试后，才将该产品方案打包交给客户。

在 Java 方面，所有 MontaVista Linux 支持的嵌入式目标机都可以运行 IBM 基于 JDK 1.2.2 独立开发的 J9 虚拟机，MontaVista 实时性的扩充和编译性能的增强使 Java 代码能在所有的 MontaVista Linux 平台上快速可靠地执行，从而可以从容应对传统嵌入式开发和移植中通常面临的平台特性不同、内存有限和程序模型不兼容的挑战。此外，VisualAge 微型版还为开发者提供了一套强大的嵌入式 Java 开发工具。

4. Android

Android 是 Google 于 2007 年 11 月 5 日宣布的基于 Linux 平台的开源手机操作系统名称，该平台由操作系统、中间件、用户界面和应用软件组成，是首个为移动终端打造的真正开放和完整的移动软件。

Google 的 Android 手机平台基于 Linux 免费开放源代码操作系统，应用是基于 Java 语言开发的。Android 基于 Apache 许可。Apache 许可模式允许开发者任意修改，分发源代码，同时，开发的新代码可以不再使用相同的许可模式，甚至可以不再开源，这为开发者带来了完全的自由，可以基于 Android 代码开发他们自己专有的平台。

Android 作为谷歌企业战略的重要组成部分，将进一步推进“随时随地为每个人提供信息”这一企业目标的实现。谷歌的目标是让移动通信不依赖于设备甚至平台。出于这个目的，Android 将补充，而不会替代谷歌长期以来奉行的移动发展战略：通过与全球各地的手机制造商和移动运营商结成合作伙伴，开发既有用又有吸引力的移动服务，并推广这些产品。

系统简介

问题：Linux 操作系统有哪些特性？Linux 操作系统的版本怎样区分？Linux 操作系统由哪些部分组成？嵌入式 Linux 应用前景如何？

重点：Linux 的特性，Linux 的系统结构，嵌入式 Linux 系统的应用前景。

内容：Linux 的特性，Linux 版本及其特点，Linux 系统的结构，嵌入式 Linux 系统及其应用前景。

简单地说，Linux 是一套免费使用和自由传播的类 UNIX 操作系统，它主要用在基于 Intel x86 系列 CPU 的计算机上。这个系统是由全世界各地的成千上万的程序员设计和实现的。其目的是建立不受任何商品化软件的版权制约的、全世界都能自由使用的 UNIX 兼容产品。

Linux 的出现，最早开始于一位名叫 Linus Torvalds 的计算机业余爱好者，当时他是芬兰赫尔辛基大学的学生。他的目的是想设计一个代替 Minix（是由一位名叫 Andrew Tannebaum 的计算机教授编写的一个操作系统示教程序）的操作系统，这个操作系统可用在 386、486 或奔腾处理器的个人计算机上，并且具有 UNIX 操作系统的全部功能，因而开始了 Linux 雏形的设计。

Linux 以它的高效性和灵活性著称。它能够在 PC 上实现全部的 UNIX 特性，具有多任务、多用户的能力。Linux 是在 GNU 公共许可权限下免费获得的，是一个符合 POSIX 标准的操作系统。Linux 操作系统软件包不仅包括完整的 Linux 操作系统，而且还包括了文本编辑器、高级语言编译器等应用软件。它还包括带有多个窗口管理器的 X-Window 图形用户界面，如同使用 Windows XP 一样，可以使用窗口、图标和菜单对系统进行操作。

Linux 之所以受到广大计算机爱好者的喜爱，主要原因有两个，一是它属于自由软件，用户不用支付任何费用就可以获得它和它的源代码，并且可以根据自己的需要对它进行必要的修改，无偿使用，无约束地继续传播。另一个原因是，它具有 UNIX 的全部功能，任何使用 UNIX 操作系统或想要学习 UNIX 操作系统的人都可以从 Linux 中获益。

5.6.1 Linux 的特性

任务：了解 Linux 操作系统的主要特性。

Linux 操作系统在短短的几年之内得到了非常迅猛的发展，这与 Linux 具有的良好特性是分不开的。Linux 包含了 UNIX 的全部功能和特性。简单地说，Linux 具有以下主要特性。

（1）开放性

开放性是指系统遵循世界标准规范，特别是遵循开放系统互连（Open System Interconnection，OSI）国际标准。凡遵循国际标准所开发的硬件和软件，都能彼此兼容，可方便地实现互连。

（2）多用户

多用户是指系统资源可以被不同用户各自拥有使用，即每个用户对自己的资源（如文件、设备）有特定的权限，互不影响。Linux 和 UNIX 都具有多用户的特性。

(3) 多任务

多任务是现代计算机的一个最主要的特点。它是指计算机同时执行多个程序,而且各个程序的运行互相独立。Linux 系统调度每一个进程平等地访问微处理器。由于 CPU 的处理速度非常快,其结果是启动的应用程序看起来好像在并行运行。事实上,从处理器执行一个应用程序中的一组指令到 Linux 调度微处理器再次运行这个程序之间只有很短的时间延迟,用户是感觉不出来的。

(4) 良好的用户界面

Linux 向用户提供了两种界面:用户界面和系统调用。Linux 的传统用户界面是基于文本的命令行界面,即 Shell,它既可以联机使用,又可以存在文件上脱机使用。Shell 有很强的程序设计能力,用户可方便地用它编制程序,从而为用户扩充系统功能提供了更高级的手段。可编程 Shell 是指将多条命令组合在一起,形成一个 Shell 程序,这个程序可以单独运行,也可以与其他程序同时运行。

系统调用是给用户提供编程时使用的界面。用户可以在编程时直接使用系统提供的系统调用命令。系统通过这个界面为用户程序提供低级、高效率的服务。

Linux 还为用户提供了图形用户界面。它利用鼠标、菜单、窗口、滚动条等设施,给用户呈现一个直观、易操作、交互性强的友好的图形化界面。

(5) 设备独立性

设备独立性是指操作系统把所有外部设备统一当成文件来看待,只要安装它们的驱动程序,任何用户都可以像使用文件一样,操纵、使用这些设备,而不必知道它们的具体存在形式。

Linux 是具有设备独立性的操作系统,它的内核具有高度适应能力,随着更多的程序员加入 Linux 编程,会有更多硬件设备加入到各种 Linux 内核和发行版本中。另外,由于用户可以免费得到 Linux 的内核源代码,因此,用户可以修改内核源代码,以便适应新增加的外部设备。

(6) 丰富的网络功能

完善的内置网络是 Linux 的一大特点。Linux 在通信和网络功能方面优于其他操作系统。其他操作系统不包含如此紧密地和内核结合在一起的连接网络的能力,也没有内置这些联网特性的灵活性。而 Linux 为用户提供了完善的、强大的网络功能。

① 支持 Internet。Linux 免费提供了大量支持 Internet 的软件,Internet 是在 UNIX 领域中建立并发展起来的,在这方面使用 Linux 是相当方便的,用户能用 Linux 与世界上的其他人通过 Internet 进行通信。

② 文件传输。用户能通过一些 Linux 命令完成内部信息或文件的传输。

③ 远程访问。Linux 不仅允许进行文件和程序的传输,它还为系统管理员和技术人员提供了访问其他系统的窗口。通过这种远程访问的功能,一位技术人员能够有效地为多个系统服务,即使那些系统位于相距很远的地方。

(7) 可靠的系统安全保障

Linux 采取了许多安全技术措施,包括对读、写进行权限控制,带保护的子系统,审计跟踪,核心授权等,这为网络多用户环境中的用户提供了必要的安全保障。

(8) 良好的可移植性

可移植性是指将操作系统从一个平台转移到另一个平台使它仍然能按其自身的方式运行的能力。

Linux 是一种可移植的操作系统,能够在从微型计算机到大型计算机的任何环境中和

任何平台上运行。可移植性为运行 Linux 的不同计算机平台与其他任何机器进行准确而有效的通信提供了手段，不需要另外增加特殊的和昂贵的通信接口。

5.6.2 Linux 版本及其特点

任务： 了解什么是 Linux 内核版本，什么是发行版本，掌握 Linux 内核版本号的规则。

Linux 版本分为内核版本和发行版本。

Linux 内核版本有两种：稳定版和开发版。稳定的内核具有工业级的强度，可以广泛地应用和部署。新的稳定内核相对于较旧的内核只是修正了一些 bug 或加入了一些新的驱动程序，而开发版内核由于要试验各种解决方案，所以变化很快，这两种版本是相互关联、相互循环的。

Linux 内核的版本号是有一定的规则的，版本号的格式是：主版本号.次版本号.修正号。主版本号和次版本号标志着重要的功能变动；修正号表示较小的功能变动。其中次版本号还有特定的意义：如果次版本号是偶数，则表示该内核是一个可放心使用的稳定版；如果次版本号是奇数，则表示该内核加入了一些测试的新功能，是一个内部可能存在 bug 的测试版。以 2.6.12 版本为例，2 代表主版本号，6 代表次版本号，12 代表修正号。例如，2.5.74 表示是一个测试版的内核，2.6.12 则表示是一个稳定版的内核。最新的内核源代码可以在 http://www.kernel.org 上以 tar 包或者增量补丁的形式下载。

Linux 发行版本有很多，这些发行版本把内核、源代码及相关的应用程序组织在一起发行，于是，同是 Linux，有不同的发行者(distributor)版本。国外比较著名的发行者及相应的版本有 OpenLinux(Caldera)、RedHat Linux(RedHat)、S. u. S. E Linux、Slackware(Walnut Creek Software) Debian GNU/Linux、Linux Mandrake、TurboLinux、LinuxPPC(Linux 的 PowerPC 版)等；国内著名的发行版本包括 Xteam Linux、Bluepoint Linux、红旗 Linux、Cosix Linux 等。还有很多著名的商业软件开发公司(如 Oracle、Informix、Sun)也着手开发了基于 Linux 的商业软件。

5.6.3 嵌入式 Linux 系统及其应用前景

任务： 了解嵌入 Linux 系统的应用前景。

近年来，随着计算机技术、通信技术的飞速发展，特别是互联网的迅速普及和 3C(计算机、通信、消费电子)合一的加速，微型化和专业化成为发展的新趋势，嵌入式产品成为信息产业的主流。Linux 从 1991 年问世到现在，短短的十几年时间已经发展成为功能强大、设计完善的操作系统之一；可运行在 x86、Alpha、Sparc、MIPS、PPC、Motorola、NEC、ARM 等多种硬件平台上，而且开放源代码，可以定制；可与各种传统的商业操作系统分庭抗礼。越来越多的企业和研发机构都转向嵌入式 Linux 的开发和研究上，在新兴的嵌入式操作系统领域内也获得了飞速发展。

嵌入式 Linux(Embedded Linux)是指对 Linux 经过裁剪小型化后，可固化在存储器或单片机中，应用于特定嵌入式场合的专用 Linux 操作系统。嵌入式 Linux 的开发和研究已经成为目前操作系统领域的一个热点。与其他嵌入式操作系统相比(详见表 5-1)，Linux 有其明显的优势。

表 5-1　专用嵌入式实时操作系统与嵌入式 Linux 的比较

操作系统 比较项目	专用嵌入式实时操作系统	嵌入式 Linux 操作系统
版权费	每生产一件产品需交纳一份版权费	免费
购买费用	数十万元(RMB)	免费
技术支持	由开发商独家提供有限的技术支持	全世界的自由软件开发者提供支持
网络特性	另加数十万元(RMB)购买	免费且性能优异
软件移植	难(因为是封闭系统)	易,代码开放(有许多应用软件支持)
应用产品开发周期	长,因为可参考的代码有限	短,新产品上市迅速,因为有许多公开的代码可以参考和移植
实时性能	好	需要改进,可用 PT_Linux 等模块弥补
稳定性	较好	较好,但在高性能系统中需要改进

第一,Linux 系统具有层次结构且内核完全开放。在内核代码完全开放的前提下,不同领域和不同层次的用户可以根据自己的应用需要方便地对内核进行改造,低成本地设计和开发出满足自己需要的嵌入式系统。

第二,强大的网络支持功能。Linux 诞生于因特网时代并具有 UNIX 的特性,保证了它支持所有标准因特网协议,并且可以利用 Linux 的网络协议栈将其开发成为嵌入式的 TCP/IP 网络协议栈。此外,Linux 还支持 ext2、FAT16、FAT32、romfs 等文件系统,为开发嵌入式系统应用打下了良好的基础。

第三,Linux 具备一整套工具链,容易自行建立嵌入式系统的开发环境和交叉运行环境,可以跨越嵌入式系统开发中仿真工具的障碍。Linux 也符合 IEEE POSIX. 1 标准,使应用程序具有较好的可移植性。

传统的嵌入式开发程序的调试和调试工具是用在线仿真器(ICE)实现的。它通过取代目标板的微处理器,给目标程序提供一个完整的仿真环境,完成监视和调试程序,但一般价格比较昂贵,只适合做非常底层的调试。使用嵌入式 Linux,一旦软硬件能够支持正常的串口功能,即使不用仿真器,也可以很好地进行开发和调试工作,从而可以节省一些开发费用。嵌入式 Linux 为开发者提供了一套完整的工具链(tool chain)。它利用 GNU 的 gcc 做编译器,用 gdb、kgdb、xgdb 做调试工具,能够很方便地实现从操作系统到应用软件各个级别的调试。

第四,Linux 具有广泛的硬件支持特性。Linux 目前已经能够支持 Intel、ARM、MIPS、PowerPC 等多种体系架构的处理器。

综上,由于 Linux 具有对各种设备的广泛支持性,因此,能方便地应用于机顶盒、IA 设备、PDA、掌上电脑、WAP 手机、车载盒以及工业控制等智能信息产品中。与 PC 相比,手持设备、IA 设备以及信息家电的市场容量要高得多,而 Linux 嵌入式系统强大的生命力和利用价值,使得越来越多的企业和高校对它表现出极大的研发热情。

小结

本章详细介绍了操作系统的基础知识,包括操作系统的定义、功能、特征、类型;进程和线程的基本知识;中断和中断的处理;内核的分类等,这些都是开发嵌入式系统必不可少的,并且对当今流行的开源和商业的嵌入式操作系统进行了介绍,最后对 Linux 系统的特点、组成和应用前景进行了简单介绍。

第6章

嵌入式Linux开发基础

学习目标

通过本章的学习,应该掌握:

- Linux的常用操作命令
- Linux的基本工具的使用方法
- Linux基本编程、编译和调试程序的方法
- 交叉编译工具链的搭建

系统的结构

问题： Linux 系统由哪些部分组成？
重点： Linux 系统的 4 个组成部分：内核、Shell、文件系统和实用工具。
内容： Linux 系统内核的组成，Shell 的作用和种类，文件和文件系统的相关知识，Linux 系统中常用的实用工具。

Linux 一般有 4 个主要部分：内核、Shell、文件系统和实用工具。

6.1.1　Linux 内核

任务： 了解 Linux 内核组成及各子系统的功能，了解子系统之间的关系。

内核是系统的心脏，是运行程序和管理如磁盘和打印机等硬件设备的核心程序。它从用户那里接收命令并把命令传送给内核去执行。Linux 内核主要由 5 个子系统组成：进程调度、内存管理、虚拟文件系统、网络接口、进程间通信。

1. 进程调度

进程调度(SCHED)控制进程对 CPU 的访问。当需要选择下一个进程运行时，由调度程序选择最值得运行的进程。可运行进程实际上是仅等待 CPU 资源的进程，如果某个进程在等待其他资源，则该进程是不可运行进程。Linux 使用了比较简单的基于优先级的进程调度算法选择新的进程。

2. 内存管理

内存管理(MM)允许多个进程安全地共享内存区域。Linux 的内存管理支持虚拟内存，即在计算机中运行的程序，其代码、数据、堆栈的总量可以超过实际内存的大小，操作系统只把当前使用的程序块保留在内存中，其余的程序块则保留在磁盘中。必要时，操作系统负责在磁盘和内存间交换程序块。内存管理从逻辑上分为硬件无关部分和硬件相关部分。硬件无关部分提供了进程的映射和逻辑内存的对换；硬件相关部分为内存管理硬件提供了虚拟接口。

3. 虚拟文件系统

虚拟文件系统(Virtual File System，VFS)隐藏了各种硬件的具体细节，为所有的设备提供了统一的接口，VFS 提供了多达数十种不同的文件系统。虚拟文件系统可以分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统，如 ext2、FAT 等，设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

4. 网络接口

网络接口(NET)提供了对各种网络标准的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序。网络协议部分负责实现每一种可能的网络传输协议。网络设备驱动程序负责与硬件设备通信，每一种可能的硬件设备都有相应的设备驱动程序。

5. 进程间通信

进程间通信(IPC)支持进程间的各种通信机制。

在这 5 个部分中,进程调度处于中心位置,所有其他的子系统都依赖它,因为每个子系统都需要挂起或恢复进程。在一般情况下,当一个进程等待硬件操作完成时,它会被挂起;当操作真正完成时,进程被恢复执行。例如,当一个进程通过网络发送一条消息时,网络接口需要挂起发送进程,直到硬件成功地完成消息的发送,当消息被成功地发送出去以后,网络接口给进程返回一个代码,表示操作成功或失败。其他子系统以相似的理由依赖于进程调度。

各个子系统之间的关系如图 6-1 所示。

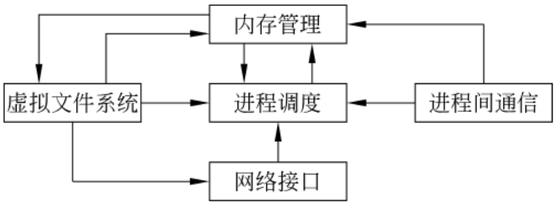


图 6-1 Linux 各子系统之间的关系图

6.1.2 Linux Shell

任务: 掌握 Shell 的作用,了解 Shell 的各种版本,Shell 命令和不同身份登录的提示符。

Shell 是系统的用户界面,提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行。

实际上 Shell 是一个命令解释器,它解释由用户输入的命令并且把它们送到内核。不仅如此,Shell 有自己的编程语言用于对命令进行编辑,它允许用户编写由 Shell 命令组成的程序。Shell 编程语言具有普通编程语言的很多特点,比如它也有循环结构和分支控制结构等,用这种编程语言编写的 Shell 程序与其他应用程序具有同样的效果。

Linux 提供了像 Microsoft Windows 那样的可视化命令输入界面——X-Window 的图形用户界面(Graphic User Interface,GUI)。它提供了很多窗口管理器,其操作就像 Windows 一样,有窗口、图标和菜单,所有的管理都是通过鼠标控制的。现在比较流行的窗口管理器是 KDE 和 GNOME。

每个 Linux 系统的用户可以拥有他自己的用户界面或 Shell,用以满足他们自己专门的 Shell 需要。

同 Linux 本身一样,Shell 也有多种不同的版本。目前主要有下列版本的 Shell。

(1) Bourne Shell:是贝尔实验室开发的。

(2) BASH:是 GNU 的 Bourne Again Shell,是 GNU 操作系统上默认的 Shell。

(3) Korn Shell:是对 Bourne Shell 的发展,大部分内容都与 Bourne Shell 兼容。

(4) C Shell:是 Sun 公司 Shell 的 BSD 版本。

Shell 有两种提示符:# 和 $ 。Linux 系统可以用两种身份登录:root 用户和一般用户。以 # 为提示符表明该终端是由 root 用户打开的,root 用户具有最高权限,因此可以输入任何可用的命令。以 $ 为提示符表明该终端是一般用户,一般用户在使用系统时是有限制的。

在 Shell 下输入相应的命令并按 Enter 键，Shell 就执行命令。如果没有此命令，Shell 会提示：command not found。Shell 命令是区分大小写的，一条命令只要有一个字母的大小写发生变化，系统就认为是一条不同的命令。输入命令、目录名或文件名开头的一个或几个字母后按 Tab 键，Shell 会在相应目录里进行匹配，自动补齐命令、目录名或文件名。还可以通过按↑、↓键来显示执行过的命令，这在重复执行某些命令时会给用户带来很大的方便。

6.1.3　Linux 文件系统

任务： 掌握 Linux 文件系统的作用，目录和路径的概念，通配符的使用，了解文件的使用权限。

文件系统是文件存放在磁盘等存储设备上的组织方法，主要体现在对文件和目录的组织上。目录提供了管理文件的一个方便而有效的途径。通过文件系统从一个目录切换到另一个目录，而且可以设置目录和文件的权限，设置文件的共享程度。

Linux 目录采用多级树状结构，如图 6-2 所示。Linux 是一个多用户系统，操作系统本身的驻留程序存放在以根目录开始的专用目录中，有时被指定为系统目录。

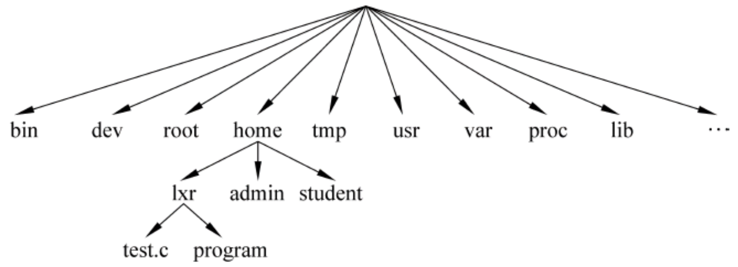


图 6-2　Linux 目录结构

在树状目录结构中，文件和目录都通过路径来表示。路径有两种表示方法：一种是从根目录开始，称为绝对路径；一种是从当前目录开始，称为相对路径。如为了标识 test. c 这个文件，可以用绝对路径/home/lxr/test. c 来表示；如果当前目录位于 home 下，则可以用 lxr/test. c 来表示。

当登录到 Linux 或打开一个终端时，会进入一个特殊目录，称为主目录。例如，root 用户登录到系统时，系统默认进入/root 目录，/root 目录就是 root 用户的主目录。主目录可以用～来表示。当前目录就是用户当前所在的目录，用户在操作时可以改变当前目录，当前目录可以用. 来表示，当前目录的父目录可以用.. 来表示。如果当前目录处于 home 下，也可以用相对路径. /lxr/text. c 来表示 test. c 文件。

在 Linux 系统中可以使用通配符 * 、? 来同时引用多个文件。通配符 * 代表文件名中任意的字符或字符串，如 abc * 表示所有以 abc 开头的文件。通配符？表示任意一个字符，如 abc？表示所有以 abc 开头的长度为 4 的文件名。

Linux 是一个多用户操作系统，为了保护用户个人的文件不被其他用户读取、修改或执行，Linux 提供文件权限机制，文件的操作权限分为读、写和执行，分别用 r、w、x 来表示。对每个文件（或目录）而言，都有 4 种不同的用户。

(1) root：系统超级用户，能够以 root 账号登录。

(2) owner：实际拥有文件(或目录)的用户。

(3) group：用户所在组的成员。

(4) other：以上 3 类用户之外的所有其他用户。

其中，root 用户自动拥有读、写和执行所有文件的操作权限，而其他 3 种用户的操作权限可以分别授予或撤销。对应于此，每个文件为后 3 种用户建立了一组 9 位的权限标识，分别赋予文件所有者、用户组和其他用户对该文件的权限。

可以用 ls -l 命令显示文件的权限，比如在 root 目录下使用该命令，部分内容如下：

```
-rw-------1 root root       1649 2008-09-28 anaconda-ks.cfg
-rw-r--r--1 root root       57590 2008-09-28 install.log
-rw-r--r--1 root root       5525 2008-09-28 install.log.syslog
-rw-r--r--1 root root       86 03-29 20:36 jieguo~
-rw-r--r--1 root root       538 2008-10-11 monitor.c~
drwxr-xr-x 4 root root      4096 04-19 23:03 mydir
-rw-r--r--1 root root       9013 12-21 22:31 nautilus-debug-log.txt
```

从第 2 列到第 10 列的 9 个字符，每 3 个一组，表示用户对文件的操作权限，-表示无此权限。例如，install.log 的文件操作权限为 rw-r--r--，第一组的 rw-表示文件拥有者对文件有读、写权限，第二组的 r--和第三组的 r--表示组用户和其他用户对文件只有读权限，而没有写和执行的权限。

内核、Shell 和文件结构一起形成了基本的操作系统结构。它们使得用户可以运行程序，管理文件以及使用系统。此外，Linux 操作系统还有许多被称为实用工具的程序，辅助用户完成一些特定的任务。

6.1.4 Linux 实用工具

任务：了解 Linux 中常用的实用工具。

标准的 Linux 系统都有一套叫做实用工具的程序，它们是专门的程序，如编辑器、执行标准的计算操作等。用户也可以创建自己的工具。

实用工具可分为以下 3 类。

(1) 编辑器：用于编辑文件。

(2) 过滤器：用于接收数据并过滤数据。

(3) 交互程序：允许用户发送信息或接收来自其他用户的信息。

Linux 的编辑器主要有 Ed、Ex、Vi 和 Emacs。Ed 和 Ex 是行编辑器，Vi 和 Emacs 是全屏幕编辑器。

Linux 的过滤器(Filter)用于读取从用户文件或其他地方的输入，检查和处理数据，然后输出结果。从这个意义上说，它们过滤了经过它们的数据。

交互程序是用户与机器的信息接口。Linux 是一个多用户系统，它必须和所有用户保持联系。信息可以由系统上的不同用户发送或接收。信息的发送有两种方式，一种方式是与其他用户一对一地链接进行通信，另一种是一个用户与多个用户同时链接进行通信，即所谓的广播式通信。

常用命令

问题：Linux 系统中包含哪些常用命令？它们该怎样使用？
重点：Linux 常用命令的使用。
内容：Linux 常用命令，包括磁盘管理命令、文件操作命令、联机帮助命令。

要使用 Linux 进行嵌入式系统的开发，首先应该掌握 Linux 常用的一些命令，以便进行程序的编写、调试和运行。本节将介绍一些常用的 Linux 命令。

6.2.1　磁盘管理命令

任务：掌握常用的磁盘管理命令的使用方法，包括 pwd、cd、ls、mkdir 等命令。

1. 查看当前目录命令 pwd

格式：pwd

功能说明：显示当前的工作目录。

范例：

显示当前的工作目录，结果如下：

```
[root@ localhost ~ ]# pwd
/root
```

表示当前目录是 root 目录。

2. 改变当前目录命令 cd

格式：cd <dirName>

功能说明：变换工作目录至 dirName。其中 driName 可以表示为绝对路径或相对路径。若目录名称省略，则变换至使用者的 home directory（也就是登录 login 时所在的目录）。另外，～也表示 home directory 的意思。

范例：

(1) 转换工作目录到/usr/bin/：cd /usr/bin

(2) 回到自己的 home directory：cd～

操作结果如下：

```
[root@ localhost ~ ]# cd /usr/bin
[root@ localhost bin]# pwd
/usr/bin
[root@ localhost bin]# cd ~
[root@ localhost ~ ]# pwd
/root
```

3. 显示目录或文件信息命令 ls

格式：ls [-alrtAFR] dirName

功能说明：列出指定目录下的文件、目录或子目录。

常用选项说明：

(1) -a　显示所有的文件和目录。

(2) -l　以长格式显示文件信息。

(3) -r　将目录以相反次序显示(原定按英文字母次序显示)。

(4) -t　将结果按建立时间排序，新的文件或目录排在前面。

(5) -A　同-a，但不列出.(当前目录)及..(父目录)。

(6) -F　在列出的名称后加一符号，例如可执行文件加 *，目录则加/。

(7) -R　若目录下有文件，则列出目录下的所有文件。

范例：

(1) 列出当前目录下以 s 开头的所有文件，新的文件排在后面：ls -ltr s *

(2) 将/bin 目录以下所有目录及文件的详细资料列出：ls -lR /bin

注意　所有的选项都以-开头，可以同时使用多个选项，此时选项开头只使用一个-即可。下同。

操作结果如下：

```
[root@localhost bin]# ls -ltr s*
-rwxr-xr-x 1 root root      107 2005-09-10 svnrevertlast
-rwxr-xr-x 1 root root      1036 2005-09-10 svnchangesince
-rwxr-xr-x 1 root root      967 2005-09-10 svnaddcurrentdir
-rwxr-xr-x 1 root root      84 2005-09-10 svnlastlog
-rwxr-xr-x 1 root root      76 2005-09-10 svngettags
-rwxr-xr-x 1 root root      1618 2005-09-10 svnforwardport
-rwxr-xr-x 1 root root      1074 2006-01-19 svnversions
-rwxr-xr-x 1 root root      4113 2006-01-19 svnlastchange
-rwxr-xr-x 1 root root      2927 2006-01-19 svn-clean
-rwxr-xr-x 1 root root      1598 2006-05-22 svnbackport
-rwxr-xr-x 1 root root      9656 2006-07-12 sgmlspl
-rwxr-xr-x 3 root root      66684 2006-07-13 sz
[root@localhost bin]# ls -lR /bin
/bin:
总计  5944
-rwxr-xr-x 1 root root      6512 2007-10-29 alsacard
-rwxr-xr-x 1 root root      19804 2007-10-29 alsaunmute
-rwxr-xr-x 1 root root      5236 2007-10-16 arch
lrwxrwxrwx 1 root root      4 2008-09-28 awk -> gawk
-rwxr-xr-x 1 root root      19200 2007-10-30 basename
-rwxr-xr-x 1 root root      735144 2007-08-31 bash
-rwxr-xr-x 1 root root      23360 2007-10-30 cat
```

4. 创建目录命令 mkdir

格式：mkdir [-p] dirName

功能说明：建立名称为 dirName 的目录。

常用选项说明：

-p　确保目录名称存在，若不存在就建一个。

范例：

(1) 在当前目录下建立一个名为 mydir 的子目录：mkdir mydir

(2) 在当前目录下的 yourdir 目录中建立一个名为 Test 的子目录。若 youdir 目录不存在，则建立：mkdir -p yourdir/Test

操作结果如下：

```
[root@localhost ~]#mkdir mydir
[root@localhost ~]#mkdir -p yourdir/Test
[root@localhost ~]#ls -l
总计  73884
-rw-------1 root root      1649 2008-09-28 anaconda-ks.cfg
-rw-r--r--1 root root      75374728 2006-12-10 cross-3.3.2.tar.bz2
-rw-r--r--1 root root      57590 2008-09-28 install.log
-rw-r--r--1 root root      5525 2008-09-28 install.log.syslog
drwxr-xr-x 2 root root     4096 04-20 01:47 mydir
drwxr-xr-x 4 root root     4096 04-19 23:03 myjob
-rw-r--r--1 root root      9013 12-21 22:31 nautilus-debug-log.txt
drwxr-xr-x 4 root root     4096 02-07 10:00 workspace
drwxr-xr-x 3 root root     4096 04-20 01:47 yourdir
[root@localhost ~]#cd yourdir
[root@localhost yourdir]#ls -l
总计  8
drwxr-xr-x 2 root root     4096 04-20 01:47 Test
```

5. 删除目录命令 rmdir

格式：rmdir [-p] dirName

功能说明：删除空目录。若目录下有文件或子目录存在，则不能删除。

常用选项说明：

-p　在子目录被删除后，若该目录也成为空目录，则一并删除。

范例：

(1) 将当前目录下删除名为 mydir 的子目录：rmdir mydir

(2) 在工作目录下的 yourdir 目录中删除名为 Test 的子目录。若 Test 删除后，yourdir 目录成为空目录，则 yourdir 也被删除：rmdir -p yourdir/Test

6.2.2 文件操作命令

任务：掌握常用的文件操作命令的使用方法，包括 cat、chmod、rm 等命令。

1. 文件内容查看和连接命令 cat

格式：cat [-AbeEnstTuv] filename

功能说明：把一个文件和几个文件连接后显示在屏幕上或其他文件中(在命令后加上＞或＞＞和文件名)。

选项说明：

(1) -n 或--number　由 1 开始对所有输出行编号。

(2) -b 或--number-nonblank　和 -n 相似，只是对空行不编号。

(3) -s 或--squeeze-blank　当遇到有连续两行以上的空行时,就替换为一行空行。

(4) -v 或--show-nonprinting　显示不可打印的字符。

范例:

(1) 把文件 textfile1 的内容加上行号后输入到文件 textfile2 中: cat -n textfile1＞textfile2

(2) 把文件 textfile1 和 textfile2 的内容加上行号(空白行不加),之后将内容附加到 textfile3 中: cat -b textfile1 textfile2＞＞textfile3

(3) 把文件 textfile1 的内容显示到屏幕上: cat textfile1

2. 修改权限命令 chmod

格式: chmod [-cfvR] mode filename

功能说明:修改文件或目录的使用权限。

参数说明:

mode　权限设定字串,格式为[ugoa][[＋－＝][rwxX]...][,...],其中,u 表示文件拥有者,g 表示与该文件的拥有者属于同一个组的组用户,o 表示其他用户,a 表示同时属于这三种用户;＋表示增加权限,－表示取消权限,＝表示唯一设定权限;r 表示可读取,w 表示可写入,x 表示可执行,X 表示只有目标文件对某些用户是可执行的或该目标目标文件是目录时才追加 x 属性。

选项说明:

(1) -c　若该文件权限确实已经更改,则显示其更改动作。

(2) -f　若该文件权限无法被更改,则不显示错误信息。

(3) -v　显示权限变更的详细信息。

(4) -R　对当前目录下的所有文件与子目录进行相同的权限变更(即以递归的方式逐个变更)。

范例:

(1) 将文件 file1. txt 设为所有人都可以读取: chmod ugo＋r file1. txt 或 chmod a＋r file1. txt

(2) 将文件 file1. txt 与 file2. txt 设为该文件拥有者,与其所属同一个组的用户允许写入,其他用户不允许写入: chmod ug＋w,o-w file1. txt file2. txt

此外,chmod 也可以用八进制数来表示权限,如 chmod 777 file。

格式: chmod nnn filename

其中,nnn 为 3 个八进制数,分别表示所有者、同组用户与其他用户的权限。可以为上面的 rwx 分别赋予不同的数值: r＝4,w＝2,x＝1,若需要 rwx 权限,则可以用 7 来表示,即 7＝4＋2＋1;若需要 rw-权限,则可以用 6 表示,即 6＝4＋2;若需要 r-x 权限,则用 5 表示,即 5＝4＋1。

例如,chmod 755 filemame,则表示文件所有者具有读、写、执行权限,同组用户具有读和执行权限,其他用户具有读和执行权限。

3. 删除文件或目录命令 rm

格式: rm [-firv] 文件或目录

功能说明:删除文件或目录。执行 rm 指令可删除文件或目录,如要删除目录,则必须

加上参数-r，否则仅会删除文件。

选项说明：

(1) -f 或--force　强制删除文件或目录。

(2) -i 或--interactive　删除文件或目录之前先询问用户。

(3) -r 或-R 或--recursive　递归处理，将指定目录下的所有文件及子目录一并处理。

(4) -v 或--verbose　显示指令执行过程。

范例：

(1) 在当前目录下，删除所有 C 源程序文件，删除前逐一询问确认：rm -i *.c

(2) 将 Finished 子目录及子目录中的所有文件删除：rm -r Finished

4. 文件移动或重命名命令 mv

格式：mv [-bfiuv] 源文件或目录　目标文件或目录

功能说明：将一个文件或目录重新命名，或移至另一目录。

选项说明：

(1) -b　若需覆盖文件，则覆盖前先进行备份。

(2) -f　若目标文件或目录与现有的文件或目录重名，则直接覆盖现有的文件或目录。

(3) -i　若目标文件或目录与现有的文件或目录重名，则覆盖前先询问用户。

(4) -u　在移动或更改文件名时，若目标文件已存在，且其文件日期比源文件新，则不覆盖目标文件。

(5) -v　执行时显示详细的信息。

范例：

(1) 将文件 aaa 更名为 bbb：mv aaa bbb

(2) 将所有的 C 源程序移至 Finished 子目录中：mv -i *.c Finished/

5. 文件复制命令 cp

格式：cp [-adfipr] 源文件或目录　目标文件或目录

功能说明：将给出的文件或目录复制到另一文件或目录中。

选项说明：

(1) -a　该选项通常在复制目录时使用。它保留链接、文件属性，并递归地复制目录，其作用等于 dpr 选项的组合。

(2) -d　复制时保留链接。

(3) -f　删除已经存在的目标文件而不提示。

(4) -i　在覆盖目标文件之前将给出提示要求用户确认。按 y 键时目标文件将被覆盖。

(5) -p　除复制源文件的内容外，还将其修改时间和访问权限也复制到新文件中。

(6) -r　若给出的源文件是一个目录文件，则递归复制该目录下的所有子目录和文件，此时目标文件必须为一个目录名。

范例：

(1) 在当前目录下复制 aaa，并将其命名为 bbb：cp aaa bbb

(2) 复制当前目录下的一个文件到另一个目录中：cp jones/home/nick/clients

(3) 复制一个目录下的所有文件到一个新目录中：cp../clients/ * /home/nick/customers

(4) 递归复制一个目录及其文件到另一个目录中：cp -r ../clients/home/nick/customers

注意 目录不能复制到目录本身中。

6. 显示文件类型命令 file

格式：file [-beLvz] 文件名

功能说明：显示文件的类型。

选项说明：

(1) -b 显示文件类型，但不显示文件名称。

(2) -c 详细显示指令执行过程，便于排错或分析程序执行的情形。

(3) -L 直接显示符号连接所指向的文件的类别。

(4) -v 显示版本信息。

(5) -z 尝试去解读压缩文件的内容。

范例：

(1) 显示文件 file1.c 的文件类型：file file.c

(2) 显示文件 file1.c 的文件类型，但不显示文件名：file -b file.c

7. 查找文件命令 find

格式：find pathname -options [-print -exec-ok...]

功能说明：从指定的起始目录开始，递归地搜索其各个子目录，查找满足条件的文件并对其进行相关的操作。

参数说明：

(1) pathname 所查找的目录路径。

(2) print 将匹配的文件输出到标准输出。

(3) exec 对匹配的文件执行该参数所给出的 Shell 命令。

(4) ok 和-exec 的作用相同，只是以一种更为安全的模式来执行该参数所给出的 Shell 命令，在执行每一个命令之前，都会给出提示，让用户来确定是否执行。

常用选项说明：

(1) -name 按照文件名查找文件。

(2) -mtime -n+n 按照文件的更改时间来查找文件，-n 表示文件更改时间距现在在 n 天以内，+n 表示文件更改时间距现在超过 n 天。find 命令还有-atime 和 ctime 选项，atime n 表示查找系统中最后 $n\times24$ 小时访问的文件，-ctime n 表示查找系统中最后 n 分钟被改变文件状态的文件。

(3) -depth 在查找文件时，首先查找当前目录中的文件，然后再在其子目录中查找。

(4) -fstype 查找位于某一类型文件系统中的文件，这些文件系统类型通常可以在配置文件/etc/fstab 中找到，该配置文件中包含了本系统中有关文件系统的信息。

(5) -follow 如果 find 命令遇到符号链接文件，就跟踪至链接所指向的文件。

范例：

(1) 将当前目录及其子目录下所有扩展名是.c 的文件列出来：find -name "*.c"

(2) 将当前目录及其子目录下所有普通文件列出：find -type f

(3) 将当前目录及其子目录下所有最近 20 分钟内更新过的文件列出：find -ctime 20

(4) 查找/var/logs 目录中更改时间在 7 天以前的普通文件，并在删除之前询问：find /var/logs -type f -mtime +7 -ok rm {} \;(注意命令后面的;一定要加上)

8. 文本搜索命令 grep

格式：grep [options] PATTERN [FILES]

功能说明：选择和显示文件中与给出的字符串相匹配的字符串或格式的行。

参数说明：

(1) option　选项。

option 常用选项说明：

① -c　只输出匹配行的计数。

② -I　不区分大小写(只适用于单字符)。

③ -h　查询多文件时不显示文件名。

④ -l　查询多文件时只输出包含匹配字符的文件名。

⑤ -n　显示匹配行及行号。

⑥ -s　不显示不存在或无匹配文本的错误信息。

⑦ -v　显示不包含匹配文本的所有行。

(2) PATTERN　匹配的模式，为一正则表达式，即要查找的字符串。

PATTERN 正则表达式主要参数：

① \　忽略正则表达式中特殊字符的原有含义。

② ^　匹配正则表达式的开始行。

③ $　匹配正则表达式的结束行。

④ \<　从匹配正则表达式的行开始。

⑤ \>　到匹配正则表达式的行结束。

⑥ [ ]　单个字符，如[A]即 A 符合要求。

⑦ [-]　范围，如[A-Z]，即 A、B、C 一直到 Z 都符合要求。

⑧ .　所有的单个字符。

⑨ *　所有字符，长度可以为 0。

(3) FILES　要在其中查找的文件，如果并没有指定文件名，将会搜索标准输入。

范例：

(1) 显示所有以 d 开头的文件中包含 test 的行：grep ‘test’ d*

(2) 显示在 aa、bb、cc 文件中匹配 test 的行：grep ‘test’ aa bb cc

(3) 显示所有包含每个字符串至少有 5 个连续小写字符的字符串的行：grep '[a-z]\{5\}' aa

(4) 如果 west 被匹配，则 es 就被存储到内存中，并标记为 1，然后搜索任意一个字符(.*)，这些字符后面紧跟着另外一个 es(\1)，找到就显示该行：grep 'w\(es\)t.*\1' aa

9. 建立文件链接命令 ln

格式：ln [options] source dist

功能说明：在 Linux/UNIX 文件系统中存在链接(link)，可以将其视为文件的别名，而链接又可分为两种：硬链接(hard link)与软链接(symbolic link)，硬链接是指一个链接可以有多个名称，而软链接则会产生一个特殊的文件，该文件的内容是指向另一个文件的位置。硬链接存在于同一个文件系统中，而软链接却可以跨越不同的文件系统。ln source dist 表示产生一个链接(dist)到 source，至于使用硬链接还是软链接则由参数决定。不论是硬链接还是软链接都不会将原来的文件复制一份，只会占用非常少的磁盘空间。

参数说明：

option 的格式为：[-bdfinsvF] [-S backup-suffix] [-V {numbered，existing，simple}]

选项说明：

(1) -f　链接时先将与 dist 同名的文件删除。

(2) -d　允许系统管理者硬链接自己的目录。

(3) -i　在删除与 dist 同名的文件前先进行询问。

(4) -n　在进行软链接时，将 dist 视为一般的文件。

(5) -s　进行软链接(symbolic link)。

(6) -v　在链接之前显示其文件名。

(7) -b　在链接时将会被覆盖或删除的文件进行备份。

(8) -S suffix　在备份的文件末尾都加上 suffix。

(9) -V METHOD　指定备份的方式。

(10) --help　显示辅助说明。

(11) --version　显示版本。

范例：

(1) 为文件 yy 建立一个软链接 zz：ln -s yy zz

(2) 为文件 yy 建立一个硬链接 zz：ln yy xx

6.2.3 联机帮助命令

任务：学会使用联机帮助命令获得系统帮助。

1. man

格式：man <command>

功能说明：列出 command 命令的使用方法，包括指令的选项与相关的参数说明。

范例：

列出 ls 命令的帮助信息：man ls

2. help

格式：<command> --help

功能说明：显示 Shell 命令的信息。

范例：

列出 ls 命令的帮助信息：ls --help

编辑、编译、调试工具

问题：在 Linux 下怎样编写程序、编译程序、调试程序？
重点：Linux 下常用编辑、编译、调试工具的使用。
内容：vi、emacs 编辑器的使用，gcc、make 编译工具的使用，gdb 调试工具的使用。

学习 Linux C 编程，必须掌握一些常用工具的使用方法，例如，Linux 下最常用的源码编辑工具 vi、emacs 等；若要编译源程序，则使用 gcc，当编译的软件包含很多文件时，还要使用 make 工具自动编译，但前提是要编写一个 Makefile 文件，Makefile 文件有许多编写规则，也要掌握；当程序出现问题时，可以使用 gdb 工具进行调试；当软件进行变更时，要进行版本控制，SVN、CVS 是 Linux，也是开源社区最常用的版本管理系统。

6.3.1　Linux 下 C 语言编程概述

任务：通过实例，了解 Linux C 编程使用的主要工具和编写程序的方法。

Linux C 编程，和在 Windows 下编程类似，一个程序要运行，一般要通过编辑、编译、链接等过程，如果程序运行出错，还要调试。下面将通过一个示例介绍在 Linux 下完成一个 C 程序的编写、编译、调试和运行的全过程。

1. 源程序的编辑

Linux 中最常用的编辑器有 vi(vim)和 emacs，它们功能强大，使用方便，广受编程爱好者的喜爱。例如，使用 vi(vim)编辑器或 emacs 编辑器编写如下文件，分别为 hello.h、linux.h、hello.c、linux.c 和 main.c，源代码如下：

```
/* main.c* /
#include "hello.h"
#include "linux.h"
int main(int argc,char**argv)
{
hello_print("Hello");
linux_print("Linux");
}
/* hello.h* /
#ifndef_HELLO_H
#define _HELLO_H
void hello_print(char * print_str);
#endif
/*  hello.c * /
#include <stdio.h>
#include "hello.h"
void hello_print(char * print_str)
{
    printf("This is hello print %s\n",print_str);
}
```

```
/* linux.h */
#ifndef _LINUX_H
#define _LINUX_H
void linux_print(char * print_str);
#endif
/* linux.c */
#include <stdio.h>
#include "linux.h"
void linux_print(char * print_str)
{
    printf("This is linux print %s\n",print_str);
}
```

2. 程序的编译和链接

由于这个程序比较短，可以按如下方式编译：

```
gcc -c main.c
gcc -c hello.c
gcc -c linux.c
gcc -o main main.o hello.o linux.o
```

main 就是最后的可执行文件。其运行结果如下：

```
./main
This is hello print:Hello
This is linux print:Linux
```

当 5 个文件中的任何一个被改写，那么上述的编译步骤又要重新进行。当文件很多时，这样做很麻烦。为了提高软件调试、维护的效率，可以使用 Linux 提供的 make 工具，方法是编写一个 Makefile 文件，然后使用 make 命令完成上述的全部编译过程，上例的 Makefile 文件内容如下：

```
main:main.o hello.o linux.o
    gcc -o main main.o hello.o linux.o
main.o:main.c hello.h linux.h
    gcc -c main.c
hello.o: hello.c hello.h
    gcc -c hello.c
linux.o:linux.c linux.h
    gcc -c linux.c
```

3. 程序的调试

通常编写的程序可能不会一次就运行成功，程序当中可能会出现许多想不到的错误，这时就要对程序进行调试。最常用的调试软件是 gdb，该软件提供了许多调试命令供用户使用，如 list 命令，可以显示源程序。如果要调试 main. c，可以使用如下命令：

```
gcc -g -c hello.c linux.c main.c
gcc -o main main.o hello.o linux.o
```

在编译时需要加上-g 选项，以便把调试信息加入到可执行文件中去。

输入命令 gdb main，进入调试环境，如图 6-3 所示。

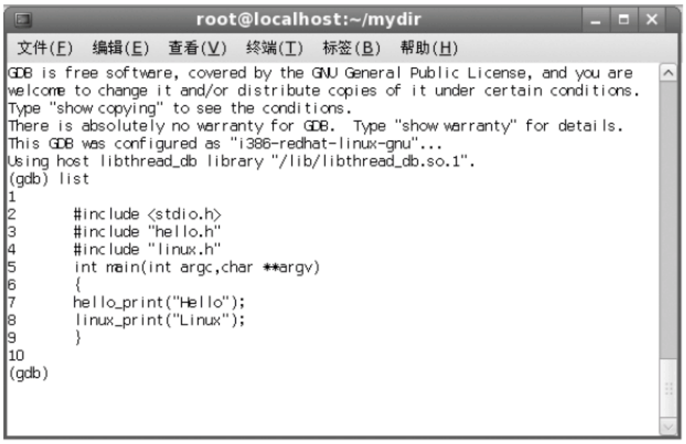


图 6-3　gdb 调试示例

6.3.2　vi 编辑器

任务： 掌握 vi 编辑器的使用方法，包括进入、退出、保存等基本操作。

vi 或 vim 是 Linux 最基本的文本编辑工具，vi 虽然没有图形界面编辑器那样单击鼠标的简单操作，但 vi 编辑器在系统管理、服务器管理中，不是图形界面的编辑器可比的。当没有安装 X-Window 桌面环境或桌面环境崩溃时，就需要使用字符模式下的编辑器 vi，vi 编辑器在创建和编辑简单文档时是最高效的工具。

1. vi 编辑器的 3 种工作模式

vi 有 3 种工作模式：命令模式、插入模式和末行模式。

（1）命令模式

在 Shell 环境中启动 vi 时，首先进入命令模式。在该模式下，用户可以输入命令，管理自己的文档，包括控制屏幕光标的移动，字符、字或行的删除、移动、复制等。此时用户从键盘输入的任何字符都作为编辑命令来解释。若用户输入的是合法的 vi 命令，则 vi 在接收用户命令之后完成相应的动作；若输入的是不合法的命令，vi 会响铃报警。需要注意的是，所输入的命令在屏幕上不显示。不管用户处于何种模式，只要用户按 Esc 键，即可使 vi 进入命令模式。

（2）插入模式

只有在插入模式下才可以进行文字输入。在命令模式下输入命令 i、附加命令 a、打开命令 o、修改命令 c、取代命令 r 或替换命令 s 都可以进入插入模式。在该模式下，用户输入的任何字符都被 vi 当做文件内容保存起来，并将其显示在屏幕上。在文本输入过程中，若想回到命令模式，按 Esc 键即可。

（3）末行模式

在命令模式下，按：键即可进入末行模式，此时 vi 会在显示窗口的最后一行显示一个：作为末行模式的提示符，等待用户输入命令。多数文件管理命令都是在此模式下执行的，如保存文件或退出 vi、寻找字符串、列出行号等。末行命令执行完后，vi 自动回到命令模式。

vi 编辑器的 3 种工作模式之间的转换如图 6-4 所示。

2. vi 的进入与退出

vi 是在 Linux 终端运行的程序，它的所有操作必须通过输入相应的命令完成。下面将介绍如何启动 vi 编辑器、保存编辑的文件以及退出 vi。

（1）进入 vi

在终端 Shell 提示符后输入 vi 和想要编辑或新建的文件名，即可进入 vi。如图 6-5 所示为输入命令 vi myfile.c 后的 vi 窗口。

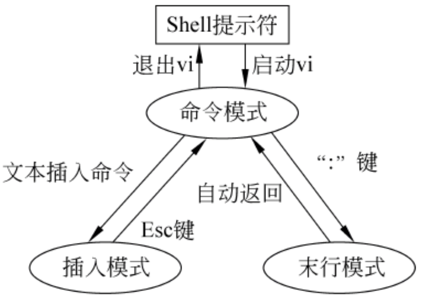


图 6-4　vi 编辑器的 3 种工作模式

图 6-5　vi 编辑器窗口

进入 vi 之后，首先进入命令模式。光标停在屏幕第一行第一列上，其余各行首均有一个～符号，表示该行为空行。最后一行称为状态行，显示当前正在编辑的文件名及其状态。在本例中，myfile.c 是一个新建文件。如果该文件已经存在，输入上述命令后则会显示出该文件的内容。

（2）保存文件和退出 vi

当编辑完文件，准备退出 vi 返回到 Shell 时，可以使用以下几种方法保存和退出 vi。

① 在命令模式下：连续按两次 Z 键，若当前编辑的文件曾被修改过，则 vi 保存该文件后退出，返回到 Shell；若当前文件没有被修改过，则直接退出。

② 在末行模式下：用以下命令保存文件。

w　vi 保存当前编辑的文件而不退出 vi，继续等待用户输入命令。

w＜newfile＞　把当前文件的内容保存到指定的文件 newfile 中，而原有文件保持不变，相当于 Windows 系统中的“另存为”命令；若 newfile 文件已经存在，则提示：file exists (use! to override)，即如果要替换原有文件，需要使用！。

w！＜newfile＞　把当前文件的内容保存到指定的文件 newfile 中，如果 newfile 已经存在，则覆盖原有内容。

③ 使用下面的方法可以退出 vi。

q　不保存文件退出 vi。若文件修改过，则提示：no write since last chang(use! to overrides，即提示使用！放弃保存。

q！　放弃对文件所做的修改，直接退出 vi 返回到 Shell。

wq　vi 先保存文件，然后退出 vi 返回到 Shell。

3. 其他命令

前面介绍了 vi 的几个基本操作，表 6-1 和表 6-2 列出了 vi 的一些常用操作。这些操作

都是在命令模式下进行的。

表 6-1 删除操作

命 令	作 用	命 令	作 用
x	删除光标所在的字符	D	同 d$
dw	删除光标所在的单词	dd	删除当前行
d$	删除自光标位置至行尾的所有字符		

表 6-2 改变与替换操作

命 令	作 用	命 令	作 用
r	替换光标所在的字符	cb	替换光标所在的前一个字符
R	替换字符序列	c$	替换自光标位置至行尾的所有字符
cw	替换一个单词	C	同 c$
ce	同 cw	cc	替换当前行

6.3.3 emacs 编辑器

任务：了解 emacs 工具的使用方法。

emacs 是 Linux 下一个功能强大的图形化文本编辑软件，可以用来编写 C 源程序。与 vi 相比，它的一个显著特点是可以使用鼠标进行大部分的操作。在安装 Linux 时若没有选择安装 emacs，可以手动安装。

1. emacs 的安装

首先下载 emacs 软件包，emacs 最新版本的源代码可以从 http://ftp.gnu.org/pub/emacs 上获得。比如下载了 emacs-22.3.tar.gz，安装过程如下。

（1）解压

```
#tar -xvzf emacs-22.3.tar.gz
```

这时解压生成一个 emacs-22.3 的目录。

（2）配置

```
#cd emacs-22.3
#./configure
```

（3）编译和安装

```
#make
#make install
```

当安装完毕后，可以在/usr/local/bin 目录下看到 emacs 可执行文件。

（4）运行

输入 emacs 命令，即可打开 emacs，其界面如图 6-6 所示。

2. emacs 的使用

emacs 工具栏中放置了一些文本编辑中常用的操作图标，如图 6-7 所示。

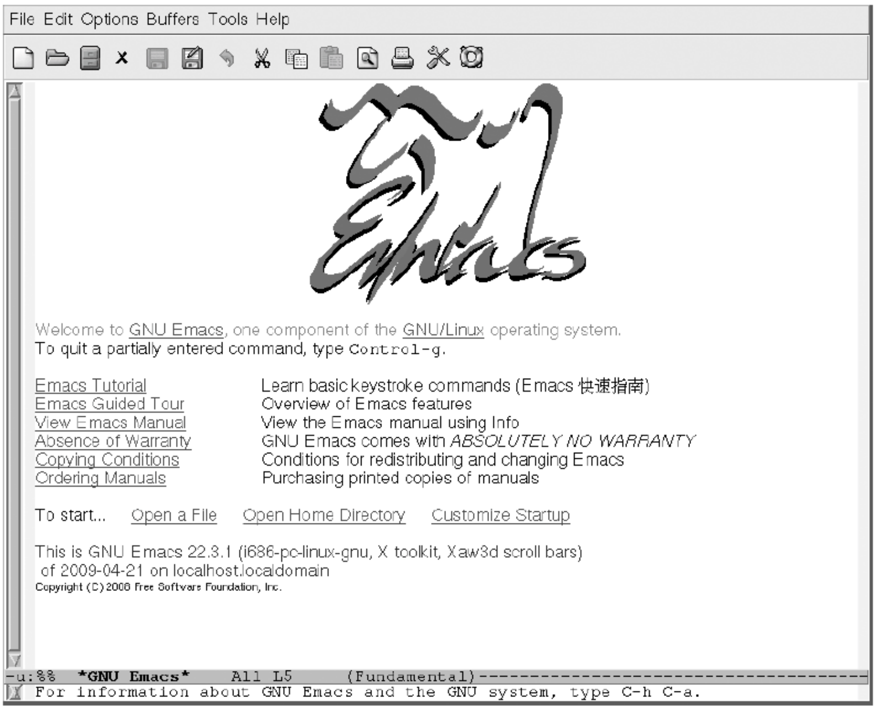


图 6-6　emacs 的启动界面

图 6-7　emacs 的工具栏

从左往右所代表的操作分别是：新建，打开文件，打开目录，关闭文件，保存，另存为，撤销，剪切，复制，粘贴，查找，打印，定制外观，帮助。

编辑区用于文本的编辑。通过上下拉动滚动条可以方便地浏览文件内容。状态栏显示文本编辑时的状态信息，如编辑的文件名、当前光标所在位置等。打开文件时，可以在命令行输入要操作的文件的路径，以便找到要操作的文件。

菜单栏中显示了 emacs 提供的所有菜单。下面列举了在编写程序时常用的一些菜单项。

(1) File 菜单项

Visit New File：新建或打开文件。若文件不存在，则新建一个文件；若文件已存在，则打开。

Open File：打开文件。

Close(current buffer)：关闭当前操作的文件。

Save(current buffer)：保存当前操作的文件。

Save As：把当前文件内容另存为其他文件。

Split Window：拆分窗口，在 emacs 中可以在不同的窗口中同时操作几个不同的文件。

Unsplit Window：取消拆分窗口。

Exit emacs：退出 emacs。

(2) Edit 菜单项

Undo：撤销上一次操作。

Cut：剪切。

Copy：复制。

Paste：粘贴。

Clear：删除。

Select All：全部选择。

Search：查找。

Go To：跳转到，可以选择跳转到相应的行、标签、文件的开头、文件的结尾。

Bookmark：标签，可以进行设置标签、重命名标签、删除标签等操作，标签是 emacs 的一个特色，在编辑比较大的文件时，设置一些标签可以方便对文件的操作。

Text Properties：设置文本的显示方式，如颜色、字体等。

(3) Options 菜单项

可以在这里设置 emacs 的一些选项，定制 emacs 的外观。

(4) Buffers 菜单项

这里列举了最近操作过的文件。

(5) Tools 菜单项

Search Files：文件查找。

Compile：编译程序。

Shell Command：执行 Shell 命令。

Debugger：调试程序。

Spell Checking：拼写检查。

Version Control：版本控制。

Read Mail：阅读邮件。

Send Mail：发送邮件。

(6) Help 菜单项

提供关于如何使用 emacs 的一些帮助信息。

3. 使用 emacs 编写、编译、运行程序

这里通过讲解 hello world 源程序从编写、编辑、编译到运行的过程，描述 emacs 的使用方法。操作步骤如下。

(1) 进入 emacs。

打开一个终端，在提示符下输入 emacs 命令，进入 emacs 环境，界面如图 6-6 所示。

(2) 建立 hello. c 文件。

单击 File 菜单中的 Visit New File 选项，则在窗口底部出现“Find file：~/”，如图 6-8 所示，然后在光标闪烁处输入新建文件的名称(假设为 hello. c)，若输入的名字与现有文件重名，则打开该文件。

(3) 编辑 hello. c 文件。

输入文件名后，便可以编辑程序了，编辑界面如图 6-9 所示。编辑完成后，单击“保存”按钮，保存该文件。

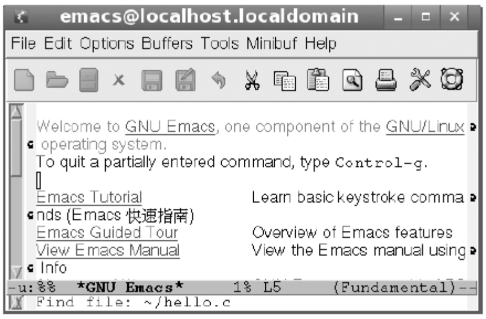


图 6-8　新建文件

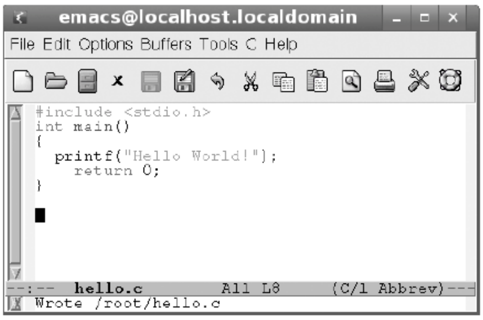


图 6-9　编辑 C 程序文件

(4) 编译 hello. c 文件。

单击 Tools 菜单中的 Compile 选项，编译源文件，在窗口底部出现命令提示“Compile command:”，默认命令为“make -k”，将其删除，输入“gcc -o hello hello. c”命令，界面如图 6-10 所示。

(5) 运行 hello 程序。

若编译无误，单击 Tools 菜单中的 Shell Command 选项，在“Shell command:”命令提示符后面输入可执行文件的名称“. /hello”，如图 6-11 所示。

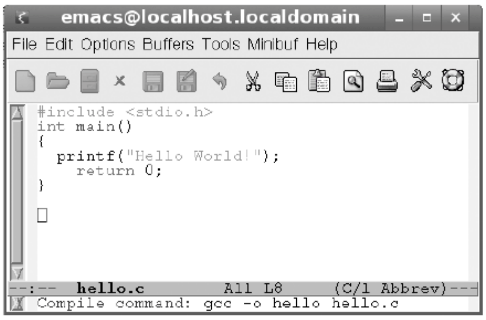


图 6-10　编译 C 程序

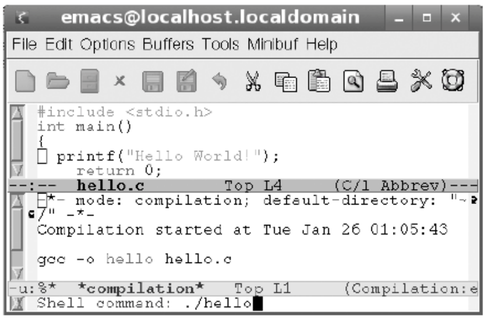


图 6-11　运行 C 程序

(6) 运行程序，结果如图 6-12 所示。

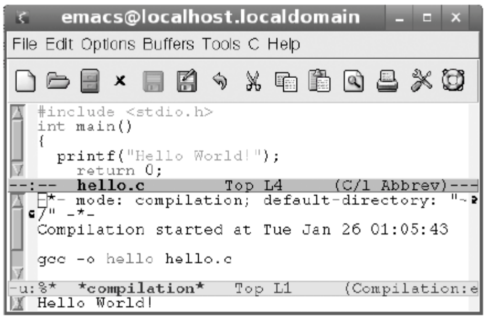


图 6-12　C 程序运行结果

6.3.4　gcc 编译工具

任务：学会使用 gcc 对源程序进行编译。

在 Linux 下开发应用程序时，在大多数情况下使用的是 C 语言，因此几乎每一位 Linux 程序员都必须掌握和灵活使用一种 C 编译器。Linux 下最常用的编译器是 gcc（GUN Compiler Collection），它是 GUN 项目中符合 ANSIC 标准的编译器。gcc 不仅功能强大，结构也非常灵活，它不仅能够编译用 C、C++ 语言编写的程序，还可以编译用 Java、Fortran、Pascal 和 Ada 等语言编写的程序。

gcc 是 Linux 平台编译器的事实标准，除了功能强大、简单灵活外，还可以支持各种不同的硬件平台。目前，gcc 支持的体系结构有几十种，常见的有 Intel x86 系列、arm、PowerPC 等。它既支持基于宿主的开发，也支持交叉编译。

1. 程序的编译过程

在使用 gcc 编译程序时，编译过程分可为 4 个阶段：预处理（preprocessing）、编译（compiling）、汇编（assembling）和链接（linking）。

Linux 程序员可以根据自己的需要使 gcc 在编译的任何阶段停止，以便检查或使用编译器在该阶段的输出信息，或对最后生成的二进制文件进行控制，以便通过加入不同数量和种类的调试代码来为今后的调试做准备。

在功能上，预处理、编译、汇编是 3 个不同的阶段，但 gcc 在实际操作时可以把这 3 个步骤合并为一个步骤来执行。下面通过实例来介绍如何生成各个阶段的代码。

在预处理阶段，输入的是 C 语言的源文件，通常为 *.c 或 *.C，它们一般带有 *.h 之类的头文件。这个阶段主要处理源文件中的 #ifdef、#include 和 #define 等预处理命令。该阶段会带有一个中间文件 *.i，但在实际工作中一般不用专门生成这种文件，若必须生成这种文件，可以使用下面的命令：

```
gcc -E test.c -o test.i
```

它通过对源文件 test.c 使用 E 选项来生成中间文件 test.i。

在编译阶段，输入的是中间文件 *.i，编译后生成汇编语言文件 *.s。这个阶段对应的 gcc 命令如下：

```
gcc -S test.i -o test.s
```

在汇编阶段，将输入的汇编文件 *.s 转换成二进制机器代码文件 *.o。这个阶段对应的 gcc 命令如下：

```
gcc -c test.s -o test.o
```

最后，在链接阶段将输入的二进制机器代码文件 *.o（与其他的机器代码文件和库文件）汇集成一个可执行的二进制代码文件。这一步骤的命令如下：

```
gcc test.o -o test
```

最终生成可执行文件 test。

对于上述过程可以简化为：

```
gcc -c test.c -o test.o
gcc test.o -o test
```

或者可以直接使用一条命令：

```
gcc test.c -o test
```

在实际开发中，使用 gcc 编译源程序时，源文件通常不止一个，这时就需要使用 gcc 编译多个源文件。例如，使用下面的命令同时编译 3 个源文件 main. c、other1. c、other2. c，最后生成一个可执行文件 test：

```
gcc -o test main.c other1.c other2.c
```

需要注意的是，在生成可执行程序时，一个程序无论是只有一个源文件还是具有多个源文件，在所有被编译和链接的源文件中必须有且只有一个 main 函数，因为 main 函数是一个程序的入口点。但如果只是把源文件编译成目标文件，因为不会进行最后一步的链接，这时 main 函数不是必需的。

2. gcc 的常用选项

在使用 gcc 编译器时，必须给出一系列必要的选项和文件名。gcc 的选项有 100 多个，其中很多参数一般是用不到的，这里只介绍其中最基本、最常用的参数。可以通过使用以下命令来详细了解所有选项：

```
man gcc
info gcc
```

gcc 最基本的用法是：

```
gcc [option] [filename]
```

其中，options 就是编译器所需要的选项，filename 用于给出相关的文件名。编译选项的含义如下。

(1) -c：只编译，不链接成可执行文件，编译器只是把输入的以. c 等为后缀的源代码文件生成以. o 为后缀的目标文件，通常用于编译不包含主程序的子程序文件。

(2) -o output_filename：确定输出文件的名称为 output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc 默认将输出的可执行文件命名为 a. out。

(3) -g：产生调试器 gdb 所必需的符号信息，要对源代码进行调试，就必须在编译程序时加入这个选项。

(4) -O：对程序进行优化编译、链接，采用这个选项，整个源代码会在编译、链接过程中被优化，这样产生的可执行文件的执行效率较高，但是，编译、链接的速度相应的会低一些。

(5) -O2：比-O 更好的优化编译、链接，当然整个编译、链接过程会更慢。

(6) -Wall：输出所有警告信息，在编译过程中 gcc 在一些可能会发生错误的地方会发出相应的警告和提示信息。提示注意这个地方是不是有什么错误。

(7) -w：关闭所有警告，建议不要使用此选项。

(8) -Idirname：将名为 dirname 的目录加入到程序头文件目录列表中，它是在预处理

阶段使用的选项。I 指 Include。

在 C 语言程序中，头文件被大量使用。在 C 程序中包含头文件有以下两种方法：

```
#include <myinc.h>
#include "myinc.h"
```

对于第一种，编译器 gcc 在系统预设包含文件目录（如 usr/include）中找到相应的头文件 myinc. h。而对于第二种，编译器 gcc 首先在当前目录中查找，若找不到，再到系统预设目录中去找。

6.3.5　gdb 调试工具

任务：学会使用 gdb 对程序进行调试。

在 Linux 应用程序开发中，最常用的调试器是 gdb。gdb 采用 GPL 授权条款，是 GUN 的计划之一，所以任何人都可以免费得到和使用它。在安装 Linux 时，如果选择了安装 gdb，gdb 就会被自动安装。gdb 和其他调试器一样，可以在程序中设置断点、查看变量值、一步一步地跟踪程序的执行过程。利用调试器的这些功能可方便地找出程序中存在的非语法错误。

1. 启动和退出 gdb

gdb 调试的对象是可执行文件，而不是程序的源代码。如果要使一个可执行文件可以被 gdb 调试，那么在使用 gcc 编译程序时需要加入-g 选项。-g 选项用于 gcc 在编译程序时加入调试信息，这样 gdb 才可以调试这个被编译的程序。

首先编写一个用于调试的测试程序 test. c。这个程序有一个名为 get_sum 的函数，它用来求 1～n 的和。为了方便查看，给该程序的各行代码进行了编号，代码如下所示：

```
1 #include <stdio.h>
2
3 int get_sum(int n);
4 {
5   int sum= 0,i;
6   for(i= 0;i<n;i++)
7   sum+= i;
8   return sum;
9 }
10
11 int main()
12 {
13   int i= 100,result;
14   result= get_sum(i);
15   printf("1+2+…+%d= %d\n",i,result);
16   return 0;
17 }
```

编译并运行该程序：

```
$ gcc -g test.c -o test
```

```
$ ./tets
1+2+…+100=4950
```

程序输出的是 4950,但本来是求 1～100 的和,应该输出 5050。程序虽然没有语法错误,但显然存在逻辑上的错误。

用 gdb 调试程序的命令格式:gdb 程序文件名

示例:

```
gdb test
```

结果如图 6-13 所示。

```
[root@localhost ~]# gdb test
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) █
```

图 6-13　利用 gdb 调试程序

在(gdb)提示符下可以输入调试命令,如果要结束调试,输入:

```
(gdb)quit
```

返回到 Linux 的提示符下。

2. 基本 gdb 命令

gdb 中提供了许多调试程序的命令,表 6-3 列出了基本的 gdb 命令。

表 6-3　基本的 gdb 命令

命　令	简　写	功　　能
file	—	装入想要调试的可执行文件
list	l	列出产生可执行文件的源代码的一部分
kill	k	终止正在调试的程序
next	n	执行一行源代码但不进入函数内部
step	s	执行一行源代码而且进入函数内部
continue	c	继续执行程序,直至下一中断或者程序结束
run	r	执行当前被调试的程序
quit	q	终止 gdb
watch	—	监视一个变量的值而不管它何时被改变
catch	—	设置捕捉点
break	b	在代码中设置断点,使程序执行到这里时被挂起
clear	c	删除一个断点,这个命令需要指定代码行或者函数名作为参数
delete	d	删除指定编号的断点,如果一次要删除多个断点,各个断点编号以空格隔开

续表

命　令	简　写	功　　能
info	—	查看断点信息
make	—	不退出 gdb，重新产生可执行文件
shell	—	不退出 gdb，执行 UNIX shell 命令
exit	—	shell 命令执行完毕，回到 gdb
print	p	打印数据内容
examine	x	打印内存内容
backtrace	bt	查看函数调用栈的所有信息

3. 用 gdb 调试程序

(1) 显示和查找程序源代码

在调试程序时，一般要查看程序的源代码。list 命令用于列出程序的源代码，它的使用格式：list [函数名][行数]

查看函数的某行或某几行代码。

示例：

① 显示 10 行代码：list

② 显示源文件 test.c 的第 5～10 行代码：list test：5，10

③ 显示 get_sum 函数的代码：list get_sum

(2) 执行程序和获得帮助

使用 gdb test 只是装入程序，程序并没有运行。如果要使程序开始运行，在 gdb 提示符下输入 run 即可。

示例：

运行程序：(gdb)run

如果想要详细了解 gdb 某个命令的使用方法，可以使用 help 命令。

示例：

① 列出 list 命令的帮助信息：(gdb) help list

② 列出 gdb 的所有命令的帮助信息：(gdb) help all

(3) 设置断点

在调试程序时，往往需要在程序运行到某行、某个函数或某个条件发生时使其暂停下来，然后查看此时程序的状态，如各个变量的值、某个表达式的值等。为此，可以设置断点。使程序运行到某个位置时暂停下来，以便检查和分析程序。

在 gdb 中，通常使用 break 命令为程序设置断点。而指定断点时，最常用的是为某行设置断点。例如：

```
(gdb) break 7
Breakpoint 1 at 0x80483da: file test.c, line 7
```

其中第二行是设置断点的返回信息。1 表示当前设置的是第一个断点，0x80483da 是断点所在的内存地址，file test.c line 7 表明断点设在 test.c 文件的第 7 行。

然后输入 run 命令运行程序，如下：

```
(gdb) run
Starting program: /root/test
Breakpoint 1, get_sum (n=100) at test.c:7
7               sum+=i;
```

可以看到，程序运行完第 6 行的指令后就暂停了，第 7 行的代码没有执行而是被 gdb 的断点中断了。此时，可以查看各个变量和表达式的值。也可以视情况让程序一步一步地执行或直接运行到程序结束。

(4) 查看断点情况

在设置完断点之后，可以输入 info breakpoints 命令查看当前设置的断点，在 gdb 中可以设置多个断点。

示例：

```
(gdb) b 7
Breakpoint 3 at 0x80483da: file test.c, line 7.
(gdb) b 14
Breakpoint 4 at 0x8048409: file test.c, line 14.
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483da in get_sum at test.c:7
2   breakpoint     keep y   0x08048409 in main at test.c:14
```

(5) 查看变量的值

在程序运行到断点暂停执行时，往往要查看变量或表达式的值，借此了解程序的执行状态。在 gdb 中输入"print 变量名"，即可查看该变量的值。例如：

```
(gdb) b 7
Breakpoint 1 at 0x80483da: file test.c, line 7.
(gdb) run
Starting program: /root/test
Breakpoint 1, get_sum (n=100) at test.c:7
7               sum+=i;
(gdb) print i
$ 1 =0
```

在 test 程序的第 7 行设置一个断点，然后使用 run 命令开始运行程序，在执行完第 6 行语句后，程序暂停下来，并提示下一条要执行的语句是第 7 行的 sum+=i。第 6 行实际执行两条语句，一条是 i=0，另一条是判断语句 i<n，故此时输出的 i 值是 0。

(6) 恢复程序运行

当程序执行到指定的断点，查看了变量或表达式的值后，可以让程序继续执行。当程序被暂停后，可以使用 continue、next、setp 语句来使程序继续执行。

① continue：执行到下一暂停点或程序结束。

② next：执行一行源代码但不进入函数内部。

③ step：执行一行源代码而且进入函数内部。

示例：

```
(gdb) list 1,17
1       #include<stdio.h>
2
3       int get_sum(int n)
4       {
5               int sum=0,i;
6               for(i=0;i<n;i++)
7               sum+=i;
8               return sum;
9       }
10
11      int main()
12      {
13              int i=100,result;
14              result=get_sum(i);
15              printf("1+2+...+%d=%d\n",i,result);
16              return 0;
17      }
(gdb) break 13
Breakpoint 1 at 0x8048402: file test.c, line 13.
(gdb) run
Starting program: /root/test

Breakpoint 1, main () at test.c:13
13              int i=100,result;
(gdb) continue
Continuing.
1+2+...+100=4950
Program exited normally.
(gdb) run
Starting program: /root/test
Breakpoint 1, main () at test.c:13
13              int i=100,result;
(gdb) next
14              result=get_sum(i);
(gdb) next
15              printf("1+2+...+%d=%d\n",i,result);
```

(7) 删除断点

使用 clear 命令或 delete 命令可以删除断点。命令格式如下。

① clear：删除程序中所有断点。

② clear 行号：删除此行的断点。

③ clear 函数名：删除该函数的断点。

④ delete 断点编号：删除指定编号的断点。如果一次要删除多个断点，各个断点编号以空格隔开。

示例：

```
(gdb) b 6
Breakpoint 1 at 0x80483d1: file test.c, line 6.
```

```
(gdb) b 7
Breakpoint 2 at 0x80483da: file test.c, line 7.
(gdb) b 8
Breakpoint 3 at 0x80483ec: file test.c, line 8.
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483d1 in get_sum at test.c:6
2   breakpoint     keep y   0x080483da in get_sum at test.c:7
3   breakpoint     keep y   0x080483ec in get_sum at test.c:8
(gdb) clear 6
Deleted breakpoint 1
(gdb) info breakpoints
Num Type           Disp Enb Address    What
2   breakpoint     keep y   0x080483da in get_sum at test.c:7
3   breakpoint     keep y   0x080483ec in get_sum at test.c:8
(gdb) delete 2 3
(gdb) info breakpoints
No breakpoints or watchpoints.
```

6.3.6 make 的使用和 Makefile 文件的编写

任务：学会使用 make 工具对工程进行管理，学会编写 Makefile 文件。

在 Linux 中，有一个用来维护程序模块关系和生成可执行程序的工具——make。它可以根据程序模块的修改情况重新编译、链接生成中间代码或最终的可执行程序。执行 make 命令，需要一个名为 Makefile 的文本文件，这个文件定义了整个项目的编译规则。本小节主要介绍 make 命令的使用和如何编写简洁、高效的 Makefile 文件。

1. Makefile 文件的基本构成

下面是一个简单的 Makefile 文件，通过它可以了解 Makefile 文件的组成和运行过程。

```
main:main.o module1.o module2.o
    gcc main.o module1.o module2.o -o main
main.o:main.c head1.h head2.h common_head.h
    gcc -c main.c
module1.o:module1.c head1.h
    gcc -c module1.c
module2.o: module2.c head2.h
    gcc -c module2.c
#this is a makefile
```

Makefile 文件的基本单元是规则，一条规则指定一个或多个目标文件，目标文件后面是编译生成该目标文件所依赖的文件或模块，最后是生成或更新目标文件所使用的命令。规则的格式：目标文件列表　分隔符　依赖文件列表[：命令]

　　　　　　　　[命令]

　　　　　　　　[命令]

其中，[]中的内容是可选的。

在上面的 Makefile 文件中，第 1、2 行就构成了一条规则。目标文件列表中只有一个目

标文件 main。main 后面的冒号(:)是分隔符,一般分隔符都是冒号。依赖文件列表是 main.o module1.o module2.o,也就是为了生成可执行程序 main,需要先生成这些依赖文件。这些依赖文件是以.o 结尾的,说明它们只是经过编译和汇编,没有进行过链接的中间代码。

第 2 行是生成目标文件所要使用的命令。需要特别注意的是,命令需要以一个 Tab 键开头,也就是说,如果某一行以 Tab 键开头,make 就认为这一行是命令。第一行用于指明模块间的依赖关系,不是命令,不能以 Tab 键开头,同样,第 3、5、7 行也不是命令。第 3、4 行,第 5、6 行,第 7、8 行也分别构成了一条规则。

最后一行以#开头,是注释行,main 命令在执行 Makefile 文件时不会解析这行内容。

2. make 解释执行 Makefile 文件的过程

当 Makefile 文件编写完成后,可以使用 make 命令来解释和执行其中的命令。在 make 命令开始执行后,首先从 Makefile 文件中获取模块间的依赖关系,判断哪些文件过时了,根据这些信息确定哪些文件需要重新编译,然后使用 Makefile 中的编译命令进行编译。所谓过时,是指一个文件生成后,用来生成该文件的源文件或头文件被修改了,导致生成该文件所需要的源文件或头文件的修改时间比生成该文件的时间晚。

假设上面的 Makefile 文件和 Makefile 文件中所涉及的源文件、头文件都在当前目录下,执行 make 命令后就开始自动编译。编译的过程如下。

(1) 在当前目录下寻找名为 Makefile 或 makefile 的文件。

(2) 在当前目录下寻找第 1 行中的目标文件 main,发现没有,就去寻找生成 main 文件所依赖的文件,即 main.o、module1.o、module2.o,发现也没有。

(3) 跳过第 2 行的编译命令,定位到第 3 行,第 3 行中的目标文件 main.o 也没有,但它所依赖的源文件和头文件在当前目录下都被找到,执行第 4 行的命令,生成 main.o 文件。

(4) 定位到第 5 行,发现目标文件 module1.o 也没有,但它所依赖的 module1.c 和 head1.h 在当前目录下能找到,执行第 6 行的命令,生成 module1.o。

(5) 以此类推,生成 module2.o。

(6) 定位到最后一行,发现是注释命令,不予处理。

(7) make 回溯到第 1 行,此时依赖文件都已经生成,于是执行第 2 行的编译命令,最后生成目标文件 main。

3. Makefile 文件的构成

一个完整的 Makefile 文件由 5 部分构成:显示规则、隐含规则、变量定义、文件指示和注释。

(1) 显示规则

一条显示规则指明了目标文件、目标文件的依赖文件、生成或更新目标文件所使用的命令。有些规则没有命令,这样的规则只描述了文件之间的依赖关系。

(2) 隐含规则

make 有自动推导功能,可以根据目标文件(典型的是根据文件名的后缀)自动推导出规则,这样可以比较简略地书写规则。例如,在 Makefile 文件中有一个规则:

```
module1.o: head1.h
```

make 根据目标文件名 module1. o 的后缀. o，自动产生目标的依赖文件 module1. c 和生成目标所使用的命令 gcc -c module1. c -o module1. o，它等价于：

```
module1.o:module1.c head1.h
    gcc -c module1.c
```

因此前面整个 Makefile 文件可以简写为：

```
main:main.o module1.o module2.o
    gcc main.o module1.o module2.o -o main
main.o: head1.h head2.h common_head.h
module1.o: head1.h
module2.o: head2.h
#this is a makefile
```

(3) 变量定义

在 Makefile 中可以定义一系列的变量，变量一般都是字符串，当 Makefile 被执行时，其中的变量会被扩展到相应的引用位置上，类似于 C 语言中的宏。

变量的一般定义形式是：变量名　赋值符　变量值

变量名习惯上只使用字母、数字和下划线，并且不以数字开头。当然也可以是其他字符，但不能使用：、#、=和空格。变量名是区分大小写的，比如变量 var 和 Var 是两个不同的变量。变量值是一个文本字符串。在含有变量的 Makefile 文件中，make 执行时把变量名出现的地方用对应的变量值来替换。赋值符主要有 4 个：=、:=、+=、?=。

当定义了一个变量之后，就可以在 Makefile 文件中使用这个变量。变量的引用方式是 $(变量名)或 ${变量名}。例如，$(foo)或者 ${foo}就是取变量 foo 的值。

在 Makefile 文件中，有两种类型的变量：递归展开变量和立即展开变量。通过=赋值的变量是递归展开变量，通过 :=赋值的变量是立即展开变量。

示例：

```
foo=$ (bar)
bar=$ (ugh)
ugh=huh
all:
    echo $ (foo)
```

执行 make 将会输出 huh。整个变量的替换过程为：在 make 执行 echo 命令时，首先 $(foo)被替换为 $(bar)，接下来 $(bar)被替换为 $(ugh)，最后 $(ugh)被替换为 huh。整个替换过程是在执行 echo $(foo)时完成的。

这种定义方式的好处是：在变量未定义时就可以使用该变量。例如，在 foo= $(bar)中，提前引用了变量 bar。如果变量 bar 在整个 Makefile 文件中都没有定义，则 $(bar)的值为空。这种定义的缺点是可能造成死循环。例如，CFLAGS= $(CFLAGS)-O，就会导致死循环。

使用赋值符“:=”赋值的变量是立即展开变量。

示例：

```
x:=$ (foo)
```

```
y:=$ (x) bar
x:=later
```

它等价于：

```
y:=foo bar
x:=later
```

这种类型的变量在定义时立即展开，而不是在引用该变量时才展开。例如：

```
CFLAGS:=$ (include_dirs)-O
include_dirs:=-lfoo -lbar
```

CFLAGS 的值是-O，而不是-lfoo -lbar -O。因为 CFLAGS 在定义时立即展开，而此时的变量 include_dirs 还未定义，那么 $ (include_dirs)的值为空。

?＝被称为条件赋值符，表示只有此变量在之前没有赋值的情况下才会对这个变量进行赋值。例如，FOO?＝bar，表示如果变量 FOO 在之前没有定义，就给它赋值为 bar。

＋＝是追加赋值符，用来实现对一个变量值的追加，这是非常有用的。通常在定义变量时，给它赋一个基本值，而后根据情况随时对其值进行追加。例如，objects＋＝another. o，表示将字符串"anther. o"添加到"objects"原有值的末尾，并用空格将其和原有值分开。

在 Makefile 文件中预定义了许多变量，可以直接使用。在隐含规则中通常会使用预定义变量，常用的预定义变量如表 6-4 所示。

表 6-4　常用的预定义变量

宏　名	初始值	说　　明
CC	cc	默认使用的编译器
CFLAGS	-o	编译器使用的选项
MAKE	make	make 命令
MAKEFLAGS	空	make 命令的选项
SHELL		默认使用的 Shell 类型
PWD		运行 make 命令时的当前目录
AR	ar	库管理命令
ARFLADS	-ruv	库管理命令选项
LIBSUFFIXE	. a	库的后缀
A	a	库的扩展名

Makefile 文件还预定义了一组变量，它们的值在 make 运行过程中可以动态改变，它们是隐含规则所必需的变量，这类变量称为自动变量。常用的自动变量有 $＠、$^、$＜，$＠代表目标文件，$^代表所有的依赖文件，$＜代表第一个依赖文件。

4. 综合举例

对前面的 Makefile 文件使用变量进行改写。

改写前：

```
main:main.o module1.o module2.o
    gcc main.o module1.o module2.o -o main
main.o:main.c head1.h head2.h common_head.h
```

```
    gcc -c main.c
module1.o:module1.c head1.h
    gcc -c module1.c
module2.o: module2.c head2.h
    gcc -c module2.c
#this is a makefile
```

改写后：

```
OBJS:=main.o module1.o module2.o
CC:=gcc
main:$ (OBJS)
    $ (CC) $^ -o $@
main.o:main.c head1.h head2.h common_head.h
    $ (CC) -c $<
module1.o:module1.c head1.h
    $ (CC) -c $<
module2.o: module2.c head2.h
    $ (CC) -c $<
#this is a makefile
```

OBJS 和 CC 是定义的变量，:= 是赋值符号，表示 main. o module1. o module2. o 是 OBJS 的值，gcc 是 CC 的值。

$(OBJS)是取变量的值，也就是将来要用“main. o module1. o module2. o”进行替换，$(CC)用“gcc”进行替换。

$^代表本规则中所有的依赖，即 main. o module1. o module2. o。

$@代表目标文件 main。

$<代表第一个依赖文件，在第 6、8、10 行分别代表 main. c、module1. c 和 module2. c。

6.3.7 版本控制

任务：了解什么是版本控制，了解版本控制常用工具 CVS 的工作模式。

在软件开发中经常遇到下面一些情况：对代码的某些部分做出了改动，但是随着时间的推移，忘记了代码改动的位置；更改代码后发现新代码不恰当，需要恢复成原来的代码；多人同时更改一个文件等，这时就要用到版本控制了。版本控制是现代软件开发中一个很关键的项目管理环节，在开发过程中，它可以跟踪和管理源代码文件变化的过程，确保由不同人员所编辑的同一文件得到更新。版本控制通过文档控制来记录程序各个模块的改动，并为每次改动编上序号，在软件开发的过程中，版本控制能解决多人并行开发和软件整合中常见的问题。

常用的版本工具是 CVS(Concurrent Versions System)，由于其简单易用，功能强大，跨平台，支持并发版本控制，而且免费，它在全球中小型软件企业中得到了广泛使用。

CVS 是一个典型的客户/服务器软件，有 UNIX 版本的 CVS、Linux 版本的 CVS 和 Windows 版本的 CVS，CVS 支持远程管理，项目组分布开发时用 CVS。CVS 的基本工作模式如图 6-14 所示。

CVS 在服务器端维护代码文档库，不同的开发者在本地机器上建立对应代码树，并利

用 CVS 保持本地代码文档同代码文档库的一致。当由于多个开发者对文件的同时修改造成本地与库中的代码文件冲突时,CVS 报告并协助解决冲突代码的合并问题。普通开发者(非管理员)对 CVS 的使用流程如图 6-15 所示。

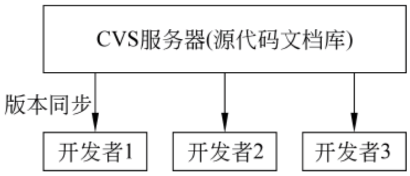


图 6-14 CVS 的基本工作模式

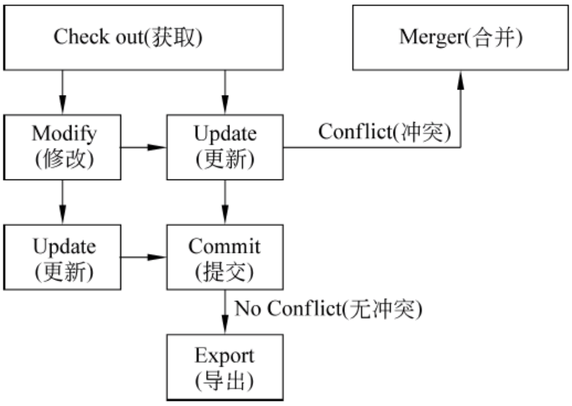


图 6-15 CVS 的使用流程

Check out 命令只需在开始建立本地代码树时使用一次,其后更新本地代码则使用 Update 命令。Update 命令将对服务器和本地代码库进行比较,并对本地代码树中过时的文件进行更新。当完成对代码的修改之后,在提交代码之前同样需要使用 Update 命令,以获取他人并行修改的代码。如果出现冲突(即对同一文件同时进行了修改),CVS 将在本地代码中把两者都保留并标记出来,要求开发者处理冲突。在冲突不存在或已解决的情况下,使用 Commit 命令将服务器代码更新为本地代码。CVS 要求为更改提供注释,并自动为更新的文件处理版本编号。当软件需要正式发布时,使用 Export 命令导出不包含 CVS 设置信息的源代码树。

编程基础

问题: 在 Linux 下怎样编写多个并发执行的程序?
重点: Linux 下的进程控制。
内容: 进程基础,Linux 下的进程控制,多线程编程入门。

Linux 是一个多用户多任务操作系统,它的一个重要特点是可以同时启动多个进程,为了让计算机在同一时间内能执行更多的任务,在进程内部又划分了许多线程。本节主要讲述 Linux 系统的进程的结构、进程的内存映像、进程的创建和退出,以及多线程编程。

6.4.1 Linux 的进程

任务: 了解 Linux 中进程的结构,理解 Linux 进程的 5 种状态。

在 Linux 操作系统中,为了唯一标识并发的进程,给每个进程分配一个进程 ID,进程 ID 是一个非负数,可以通过调用函数 getpid()获得。

Linux 中的进程由 3 部分组成：代码段、数据段和堆栈段，如图 6-16 所示。

代码段	数据段	堆栈段

图 6-16　Linux 进程组成

代码段存放程序的可执行代码。数据段存放程序的全局变量、常量和静态变量。堆栈段中的堆用于存放动态分配的内存变量，比如使用 malloc()函数分配的内存空间；堆栈段中的栈用于函数调用，其中存放着函数的参数、函数内部定义的局部变量。

Linux 进程有以下 5 种状态。

(1) 运行状态(TASK_RUNNING)

当进程正在被 CPU 执行，或已经准备就绪可由调度程序调度执行时，称该进程处于运行状态(running)。

(2) 可中断睡眠状态(TASK_INTERRUPTIBLE)

当进程处于可中断等待(睡眠)状态时，系统不会调度该进程执行。当系统产生一个中断或者释放了进程正在等待的资源，或者进程收到一个信号时，都可唤醒进程转换到运行状态。

(3) 不可中断睡眠状态(TASK_UNINTERRUPTIBLE)

除了不会因为收到信号而被唤醒，该状态与可中断睡眠状态类似。但处于该状态的进程只有被 wake_up()函数唤醒时才能转换到运行状态。该状态通常在进程需要不受干扰地等待或者所等待事件会很快发生时使用。

(4) 暂停状态(TASK_STOPPED)

当进程收到信号 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 时就会进入暂停状态。可向其发送 SIGCONT 信号让进程转换到可运行状态。进程在调试期间接收到任何信号均会进入该状态。

(5) 僵死状态(TASK_ZOMBIE)

当进程已停止运行，但其父进程还没有调用 wait()询问其状态时，称该进程处于僵死状态。为了让父进程能够获取其停止运行的信息，此时子进程的任务数据结构信息还需要保留着。一旦父进程调用 wait()取得了子进程的信息，则处于该状态进程的任务数据结构就会被释放。

6.4.2　Linux 下的进程控制

任务： 掌握 Linux 中进程控制常用的系统调用 fork、exit、exec、wait、getpid 等的使用方法。

Linux 进程控制包括创建进程、执行新程序、退出进程等。Linux 系统为了对进程进行控制，提供了一系列的系统调用，用户可以通过这些系统调用来完成创建一个新进程、终止一个进程等操作。本小节主要介绍以下几个系统调用的用法。

(1) fork：用于创建一个新进程。

(2) exit：用于终止进程。

(3) exec：用于执行一个应用程序。

(4) wait：将父进程挂起，等待子进程终止。

(5) getpid：获取当前进程的进程 ID。

1. 创建进程

一个进程可以创建另一个进程,新创建的进程称为子进程,原来的进程称为父进程,子进程还可以创建子进程,这样就形成了一个进程家族。

fork 函数是创建一个新进程的唯一方法,其函数原型如下:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

成功调用 fork 函数后,当前进程实际上已经分裂为两个进程,一个是原来的父进程,另一个是刚刚创建的子进程。子进程和父进程使用相同的代码段,子进程复制父进程的堆栈段和数据段。子进程一旦开始运行,父、子进程之间不再相互影响。

在一般情况下,函数最多有一个返回值,但 fork 函数非常特殊,它有两个返回值,一个是父进程调用 fork 函数后的返回值,该返回值是新创建的子进程的进程 ID;另一个是子进程中 fork 函数的返回值,其值为 0。如果进程创建失败,则只返回-1。下面的例子是 fork 函数的常见用法。

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    pid_t  pid;

    printf("Process Creation Study\n");
    pid=fork();
    switch(pid) {
        case 0:
            printf("Child process is running,CurPid is %d,
                ParentPid is %d\n", pid, getppid());
            break;
        case -1:
            perror("Process creation failed\n");
            break;
        default:
            printf("Parent process is running,ChildPid is %d,
                ParentPid is %d\n", pid, getpid());
            break;
    }
    exit(0);
}
```

程序的一次运行结果如下:

```
Process Creation Study
Child process is running,CurPid is 0, ParentPid is 17236
Parent process is running,ChildPid is 17237, ParentPid is 17236
```

从程序的运行结果可以看出,进程创建成功后,fork 函数返回了两次:一次返回值是

0,代表子进程在运行,通过函数 getppid 得到其父进程 ID 为 17236;另一次返回值为 17237,代表当前父进程在运行,通过函数 getpid 得到其父进程 ID 为 17236,刚好与前面得到的父进程 ID 一致。

再次运行程序,系统分配给进程的 ID 一般会发生变化。例如,再次运行该程序,结果如下:

```
Process Creation Study
Child process is running,CurPid is 0, ParentPid is 17318
Parent process is running,ChildPid is 17319, ParentPid is 17318
```

在上例中,两次运行结果均表明子进程先运行。一般来说,执行 fork 函数后是父进程先运行,还是子进程先运行是不确定的,这取决于内核所使用的调度算法。

2. 进程退出

进程退出表示进程即将结束运行。在 Linux 系统中进程退出分为正常退出和异常退出。

正常退出包括:

(1) 在 main 函数中执行 return。

(2) 调用 exit 函数。

异常退出包括:

(1) 调用 abort()函数。

(2) 进程收到某个信号,而该信号使进程终止。

不管是哪种退出方式,最终都会执行内核中的同一段代码,以关闭进程所有已打开的文件描述符,释放它所占用的内存和其他资源。

3. 执行新程序

使用 fork 函数创建子进程后,子进程通常会调用 exec 函数族来执行另外一个程序。系统调用 exec 用于执行一个可执行程序以代替当前进程的进程映像。进程一旦调用 exec()函数族,它本身就"死亡了",系统把代码段替换成新程序的代码,废弃原有的数据段和堆栈段,并为新进程分配新的数据段和堆栈段,唯一留下的就是进程号。

在 Linux 中 exec 函数族有 6 种不同的调用形式,包括 execv、execve、execl、execle、execvp、execlp。它们的用法类似,下面举一个例子加以说明。

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    pid_t  pid;
    printf("Executing new process study\n");
    pid=fork();
    switch(pid) {
        case 0:
            printf("Child process is running\n");
            execlp("ls","ls","-l",(char *)0);
            printf("execlp erro\n");          //该语句永远不会执行,除非 execlp 调用失败
```

```
                break;
            case -1:
                perror("Process creation failed\n");
                break;
            default:
                wait(&pid);
                printf("child process end\n");
                break;
        }
        exit(0);
    }
```

运行结果：

```
Executing new process study
Child process is running
总计 104
-rwxr-xr-x 1 root root 5554 03-10 23:08 forke
-rw-r--r--1 root root  127 2009-03-26 hello.c
-rw-r--r--1 root root  100 2009-03-26 hello.h
-rw-rw-r--1 root root 2132 2009-03-26 hello.o
-rw-r--r--1 root root  130 2009-03-26 linux.c
-rw-r--r--1 root root   98 2009-03-26 linux.h
-rw-r--r--1 root root   98 2009-03-26 linux.h~
-rw-rw-r--1 root root 2132 2009-03-26 linux.o
-rwxrwxr-x 1 root root 7320 2009-03-26 main
-rw-r--r--1 root root  199 2009-03-29 main.c
-rw-rw-r--1 root root 2496 2009-03-26 main.o
-rw-r--r--1 root root  201 2009-03-26 Makefile
child process end
```

在本程序中，子进程执行了 ls -l 命令，父进程等待子进程运行结束，当子进程运行结束后，父进程结束。

4. 等待进程结束

当子进程先于父进程退出时，如果父进程没有调用 wait 函数，子进程就会进入僵死状态。如果父进程调用了 wait 函数，就不会使子进程变为僵尸进程。

wait 函数的声明如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int * staloc);
```

wait 函数用于使父进程暂停执行，直到它的一个子进程结束为止。该函数的返回值是终止运行的子进程的 PID。参数 staloc 所指向的变量存放子进程的退出码，即从子进程的 main 函数返回的值或子进程中 exit 函数的参数。其用法见上例中的"wait(&pid);"语句，当子进程结束后，将返回子进程的 PID。

6.4.3 多线程编程入门

任务：掌握在 Linux 中用于创建线程的系统调用 pthread_create、pthread_exit 的方法。

一个进程可以创建多个线程,多个线程可以并发执行,线程是计算机中独立运行的最小单位,运行时占用很少的系统资源。

1. 创建线程

前面的程序实例都是单线程的。单线程的程序都是按照一定的顺序执行的,如果在主线程里创建线程,程序会在创建线程的地方产生分支,变成两个程序执行。但要注意,这和多进程不一样。子进程是通过复制父进程的地址空间来实现的;而线程与进程内的线程共享程序代码,一段代码可以同时被多个线程执行。

线程的创建是通过函数 pthread_create 来完成的,其函数原型如下:

```
#include <pthread.h>
int pthread_create(pthread_t * thread,pthread_attr_t * attr,
                   void*  (* start_routine)(void* ),void * arg)
```

参数说明:

thread:该参数是一个指针,当线程创建成功时,用来返回创建的线程 id。

attr:用于指定线程的属性,null 表示使用默认属性。

start_routine:指向线程创建后要调用的函数。这个线程调用的函数也称为线程函数。

arg:指向传递给线程函数的参数。

示例:

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<pthread.h>
void thread(void * arg)
{
    int i;
    for(i=0;i<3;i++)
    {
        printf(" new thread is running.\n");
    }
}
int main(void)
{
    pthread_t id;
    int i;
    int ret;
    ret=pthread_create(&id,NULL,(void* )thread,NULL);
    if(ret!=0)
    {
        printf("Create pthread error!\n");
        exit(1);
    }
    for(i=0;i<3;i++)
    {
        printf(" main thread is running.\n");
    }
    pthread_join(id,NULL);
```

```
    return(0);
}
```

在该例中,主线程创建了一个子线程,子线程先运行,主线程后运行,主线程通过调用"pthread_join(id,NULL);"等待子线程运行结束。注意在编译时,使用-lpthread 参数。程序运行结果如下:

```
[root@ localhost mydir]# gcc -o thread thread.c -lpthread
[root@ localhost mydir]# ./thread
  new thread is running.
  new thread is running.
  new thread is running.
  main thread is running.
  main thread is running.
  main thread is running.
```

2. 线程终止

在 Linux 中可以采用两种方式终止线程,第一种是通过 return 从线程函数返回,第二种是通过调用 pthread_exit()使线程退出。pthread_exit 在头文件 pthread. h 中声明,该函数原型如下:

```
#include<pthread.h>
void pthread_exit(void * retval);
```

有两种情况要注意:一种情况是,在主线程中,如果从 main 函数返回或是调用 exit 函数退出主线程,则整个进程将终止,此时进程中的所有线程也将终止,因此在主线程中不能过早地从 main 函数返回;另一种情况是如果主线程调用 pthread_exit 函数,则仅仅是主线程消亡,进程不会结束,进程内的其他线程也不会终止,直到所有线程结束,进程才会结束。

函数 pthread_join 用来等待一个线程的结束,该函数也在头文件 pthread. h 中声明,原型如下:

```
#include<pthread.h>
void pthread_exit(void * retval);
void pthread_join(void phtread_t th,void*  thread_return);
```

pthread_join()函数的调用者将被挂起并等待 th 线程终止,如果 thread_return 不为 NULL,则 * thread_return=retval。需要注意的是,一个线程仅允许一个线程使用 pthread_join()等待它的终止。示例程序如下:

```
##include <stdio.h>
#include <pthread.h>
void assisthread(void *  arg)
{
    printf ("I am helping to do some jobs\n");
    sleep (3);
    pthread_exit (0);
}
int main(void)
{
```

```
    pthread_t jointhid;
    int         status;
    pthread_create (&jointhid, NULL, (void * ) assisthread, NULL);
    pthread_join(jointhid, (void * ) &status);
    printf("jointhread's exit is caused %d\n", status);
    return 0;
}
```

该程序演示了子线程可以使用 pthread_exit 终止，主线程通过 pthread_join 等待辅助线程结束。运行结果如下：

```
I am helping to do some jobs
jointhread's exit is caused 0
```

从程序运行结果可以看出，pthread_join 会阻塞主线程，等待线程 jointhread 结束。pthread_exit 结束时的退出码是 0，pthread_join 得出的 status 也为 0，两者是一致的。

调试程序

问题：在 Linux 中怎样调试多个并发执行的线程程序或进程程序？
重点：使用 gdb 调试多线程程序和多进程程序的方法。
内容：在 Linux 系统中调试多线程程序和多进程程序。

6.5.1 调试多线程程序

任务：学会使用 gdb 进行多线程程序的调试。

线程有自己的寄存器，运行时堆栈或许还会有私有内存。gdb 提供了以下供调试多线程的进程的功能。

(1) 自动通告新线程。
(2) 在线程之间进行切换，提供了 thread THREADNO 命令。
(3) 查询现存线程，提供了 info threads 命令。
(4) 可以在线程内设置断点。

注意，不是所有的 gdb 版本都能对线程进行调试，如果 gdb 不支持这些命令，会显示出错信息：

```
(gdb)info threads
(gdb)thread 1
Thread ID 1 not known. Use the \ "info threads\ " command to see the IDs of currently known
threads.
```

在下面的示例中，编写程序 threadgdb. c，编译运行后会产生 3 个线程，其中一个为主线程，两个为子线程。示例程序如下：

```
#include<stdio.h>
#include<pthread.h>
```

```
#include<unistd.h>
#include<stdlib.h>
void thread1(void * arg)
{
    int i;
    for(i=0;i<5;i++)
    {
        printf(" thread1 is running.\n");
    sleep(1);
    }
}
void thread2(void * arg)
{
    int i;
    for(i=0;i<5;i++)
    {
        printf(" thread2 is running.\n");
        sleep(1);
    }
}
int main(void)
{
    pthread_t id1,id2;
    int i;
    int ret1,ret2;
    ret1=pthread_create(&id1,NULL,(void*)thread1,NULL);
    ret2=pthread_create(&id2,NULL,(void*)thread2,NULL);
    if(ret1!=0)
    {
        printf("Create pthread1 error!\n");
        exit(1);
    }
    if(ret2!=0)
    {
        printf("Create pthread2 error!\n");
        exit(1);
    }
    for(i=0;i<3;i++)
    {
        printf(" main thread is running.\n");
    }
    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
    printf(" All threads are over");
    return(0);
}
```

对其进行编译，为了加入调试信息，需要加上-g 选项：

```
[root@localhost mydir]#gcc -g -o threadgdb thread.c -lpthread
```

运行结果如下：

```
[root@ localhost mydir]# ./threadgdb
thread1 is running.
thread2 is running.
main thread is running.
main thread is running.
main thread is running.
thread1 is running.
thread2 is running.
thread1 is running.
thread2 is running.
thread1 is running.
thread2 is running.
thread1 is running.
thread2 is running.
All threads are over
```

下面使用 gdb 提供的命令对线程进行调试。

首先进入 gdb，然后设置断点位置在第 32 行，运行程序后，系统创建了一个进程，其 ID 为 21925，该进程产生了 3 个线程，编号分别为 21926、21925、21927。LWP 的含义是轻量级进程，实际上就是 Linux 中的线程。结果如下：

```
(gdb) threadgdb
(gdb) b 32
Breakpoint 3 at 0x80485f9: file thread.c, line 32.
(gdb) r
Starting program: /root/mydir/threadgdb
[Thread debugging using libthread_db enabled]
[New process 21925]
[New Thread -1209033840 (LWP 21926)]
[New Thread -1209030976 (LWP 21925)]
thread1 is running.
[New Thread -1219523696 (LWP 21927)]
[Switching to Thread -1209030976 (LWP 21925)]
Breakpoint 3, main () at thread.c:32
if(ret1!=0)
```

使用 info threads 命令，查看当前现存的线程，结果如下：

```
(gdb) info threads
4 Thread -1219523696 (LWP 21927)   0x0052db18 in clone () from /lib/libc.so.6
*  3 Thread -1209030976 (LWP 21925)   main () at thread.c:32
2 Thread -1209033840 (LWP 21926)   0x00110402 in __kernel_vsyscall ()
1 LWP 21925   main () at thread.c:32
```

1、2、3、4 是 gdb 分配的线程号，切换线程时使用该号码；带有 * 的线程为当前活动的线程。在使用 gdb 调试时，通常只有一个线程为活动线程。

使用 thread THREADNO 命令切换活动线程，比如设置线程 2 为活动线程，可以输入下面的命令，通过 info threads 命令查看，发现线程 2 确实成为了活动线程。

```
(gdb) thread 2
[Switching to thread 2 (Thread -1209033840 (LWP 21926))]#0  0x00110402 in __kernel_vsyscall
()
(gdb) info threads
4 Thread -1219523696 (LWP 21927)  0x0052db18 in clone () from /lib/libc.so.6
3 Thread -1209030976 (LWP 21925)  main () at thread.c:32
* 2 Thread -1209033840 (LWP 21926)  0x00110402 in __kernel_vsyscall ()
1 LWP 21925  main () at thread.c:32
```

下面通过使用 c 命令，继续运行程序，直到所有的线程运行结束并且退出。

```
(gdb) c
Continuing.
thread2 is running.
main thread is running.
main thread is running.
main thread is running.
thread1 is running.
thread2 is running.
thread1 is running.
thread2 is running.
thread1 is running.
thread2 is running.
thread1 is running.
thread2 is running.
[Thread -1209033840 (LWP 21926) exited]
[Thread -1219523696 (LWP 21927) exited]
All threads are over
Program exited normally.
```

6.5.2　调试多进程程序

任务：学会使用 gdb 进行多进程程序的调试。

通常使用 gdb 调试子进程的方式是在子进程里插入 sleep()语句。运行 gdb 调试这个程序，等到子进程执行 fork 函数后，得到它的进程号，然后在另外一个终端运行 gdb，用这个进程号附加上这个子进程，就可以调试子进程了。

示例：编写程序 forkgdb.c，然后对其子进程进行调试。程序源代码如下：

```
#include <stdio.h>
#include<unistd.h>
#include<stdlib.h>

int main()
{
    pid_t pid;
    int i=0;
     pid=fork();
    if(pid==0)
```

```
    {
        printf("Child Process say hello to you!\n");
          sleep(10);
        printf("Child wake up!\n");
        while(i<3){i++; printf("i=%d\n",i);}
        exit(0);
    }
    else
    {
        printf("Parent Process,Hello\n");
        wait(pid);
        printf("Parent Process is over!\n");
        return 0;
    }
}
```

编译：

```
gcc -g -o forkgdb forkgdb.c
```

调试步骤如下。

(1) 在两个终端分别运行 gdb 进行调试。

(2) 在其中一个终端运行该程序，当产生新进程后，在另一个终端输入 attach＋子进程号。

在终端 1 中当输入命令 r 时，会发现 fork 被调用之后产生了新进程，进程 ID 为 31066，并且当子进程运行到 sleeping(10)时，子进程暂时停止运行，具体内容如下：

```
[root@ localhost mydir]# gdb forkgdb
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) r
Starting program: /root/mydir/forkgdb
[Detaching after fork from child process 31066. (Try 'set detach-on-fork off'.)]
Parent Process,Hello
Child Process say hello to you!
```

当在终端 1 中发现新进程被创建了，并且子进程处于睡眠状态时，在终端 2 中输入命令 attach 31066，程序进入子进程中：

```
[root@ localhost mydir]# gdb forkgdb
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
```

```
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) attach 31066
Attaching to program: /root/mydir/forkgdb, process 31066
Reading symbols from /lib/libc.so.6...
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...
done.
Loaded symbols for /lib/ld-linux.so.2
0x00110402 in __kernel_vsyscall ()
(gdb)
```

(3) 在终端 2 的子进程中设置断点，比如在第 14 行设置断点，输入命令 b 14，然后继续运行，即输入命令 r，子进程在断点处暂停。具体命令和运行结果如下：

```
(gdb) b 14
Breakpoint 1 at 0x80484fe: file forkgdb.c, line 14.
(gdb) c
Continuing.
Breakpoint 1, main () at forkgdb.c:14
14                      while(i<3){i++; printf("i=%d\n",i);}
(gdb)
```

观察终端 1，发现并没有任何进展。

(4) 在终端 2 中逐步输入 s 命令，进行单步调试，直到子进程运行结束，退出。观察终端 1，发现了子进程的输出结果，具体命令和运行结果如下：

```
(gdb) s
15                      exit(0);
(gdb) s
Program exited normally.
```

终端 1 的运行结果在原来的基础上增加了如下内容：

```
i=1
i=2
i=3
Parent Process is over!
Program exited normally.
```

交叉编译

问题： 什么是交叉编译？开发嵌入式系统为什么需要交叉编译？怎样安装交叉编译工具链？交叉编译常用的工具有哪些？

重点： 安装交叉编译工具链。

内容： 嵌入式系统开发模型，交叉编译工具链的安装，交叉编译常用的工具，最后介绍了一个交叉编译的完整实例。

6.6.1 嵌入式系统开发模型

任务：了解嵌入式系统开发模型。

嵌入式系统的开发通常采用“宿主机-目标机”的交叉开发方式。宿主机的操作系统一般是通用的 Windows 或 Linux 系统。目标机就是嵌入式应用系统，形态和结构各异，其上运行嵌入式操作系统，与主机通过串口、以太网口、JTAG 调试口、USB 口或其他方式通信，下载运行在宿主机中编译好的代码。

开发环境建立在宿主机上，用户所有的开发工作大都在宿主机开发环境中进行，包括程序编辑、编译、链接等。生成的可执行目标代码通过串口或以太网口下载到目标机，在目标机执行时，可以把执行结果回显到宿主机上，宿主机还可以通过开发环境提供的调试工具对代码进行调试。

6.6.2 交叉编译工具链

任务：理解交叉编译技术和交叉编译工具链的概念，掌握交叉编译工具链的安装方法。

当需要从源代码编译出一个能运行在 ARM 架构上的程序时，可以采用两种方法。第一种是使用相同架构机器上的编译器，编译出运行在同一架构上的程序。在嵌入式系统领域，这是较难实现的。嵌入式系统在设计时由于考虑到功耗、体积等要求，性能往往较弱。在编译一些较大规模程序时，嵌入式系统在编译上耗费的时间是无法忍受的。因此，需要使用交叉编译技术。

1. 交叉编译技术

所谓交叉编译技术，其实是一种在一个异构平台上编译出目标平台程序的技术。从理论上来说，交叉工具链可以用在任何两种异构的系统中，例如，可以构建出 PowerPC-ARM 工具链，Sun Sparc-x86 工具链等。目前交叉工具链一般用于目标平台计算能力较弱，需要其他计算能力较强的平台帮助产生可运行软件的场合。在基于 ARM 的嵌入式系统开发中，一般会使用 x86 架构的计算机系统作为工作站，故最为常用的是 x86-arm 交叉工具链。

2. 交叉编译工具链的概念

每一个软件在编译的过程中，都要经过一系列的处理，才能从源代码变成可执行的目标代码。这一系列处理包括预编译、高级语言编译、汇编、链接及重定位。这一套流程里面用到的每个工具和相关的库组成的集合，就称为工具链(tool chain)。以 GNU 的开发工具 gcc 为例，它包括预编译器 cpp、C 编译器 gcc、汇编器 as 和链接器 ld 等。在 GNU 对工具链的定义中，还加进了一套额外的用于处理二进制包的工具包 binutils，整个工具链应该是 gcc＋binutils＋Glibc。

在一般情况下，工具链运行的环境和它产生的目标代码的环境是一致的。例如，在 Visual C++ 中编译一个程序，工具链运行在 x86 平台上，产生的也是运行在 x86 平台上的代码。但实际上，工具链产生的目标代码的运行平台是可以跟工具链运行的环境不一致的。这种产生与运行环境不一致的目标代码的工具链称为交叉工具链(cross-compiler tool chain)，使用这种工具链的编译过程对应地被称为交叉编译(cross-compile)。在 GNU 中，一般在普通工具链名称的前面加上特定的前缀，以表示是什么类型的工具链。如 x86-arm

的工具链预编译器是 arm-linux-cpp,C 编译器是 arm-linux-gcc 等。

因此,在进行交叉编译之前,首先要安装交叉编译工具链。交叉编译工具链可以直接使用制作好的工具链,也可以自己制作,对于初学者而言,一般使用制作好的工具链。例如,对于 ARM 处理器和嵌入式 Linux 2.4 版本的内核需要使用交叉编译工具的版本是 cross-2.95.3,嵌入式 Linux 2.6 内核的通常使用 cross-3.3.2 版本。这些工具可以从互联网上或者开发商处获得已经编译好的开发环境。

3. 交叉编译工具链的安装

以安装交叉编译工具 cross-3.3.2 版本为例讲解怎样安装交叉工具链。

首先,到网站上下载 cross-3.3.2.tar.bz2,然后,将其解压,将解压后的工具包安装到指定的路径,一般为/usr/local/arm/目录下,可以使用命令 mv 3.3.2/ /usr/local/arm/进行移动。最后,设置 PATH 环境变量,此时要打开/root/.bash_profile 文件,在该文件中加入下面的语句:

```
PATH=$PATH:/usr/local/arm/3.3.2/bin
```

保存退出后,再执行命令:

```
#source ~/.bash_profile
```

使环境变量生效。接下来就可以使用这个交叉编译工具进行交叉编译工作了。

在交叉工具链中有许多常用的工具,其介绍如表 6-5 所示。

表 6-5　交叉编译常用工具介绍

名　称	归　属	说　　明
arm-linux-as	binutils	编译 ARM 汇编程序
arm-linux-ar	binutils	把多个.o 文件合并成一个.o 文件或静态库(.a)
arm-linuxran-lib	binutils	为库文件建立索引,相当于 arm-linux-ar-s
arm-linux-ld	binutils	链接器,把多个.o 文件或库文件链接成一个可执行文件
arm-linux-objdump	binutils	查看目标文件(.o)和库(.a)的信息
arm-linux-copy	binutils	转化可执行文件的格式
arm-linux-strip	binutils	去掉 elf 可执行文件的信息
arm-linux-readelt	binutils	读 elf 可执行文件的信息
arm-linux-gcc	gcc	编译以.c 或.s 结尾的 C 程序或汇编程序
arm-linux-g++	gcc	编译 C++ 程序

6.6.3　交叉编译实例

任务: 掌握交叉编译的全过程。

下面以 6.3.1 小节的程序为例,使用交叉编译工具进行编译,源代码如下:

```
/* main.c* /
#include "hello.h"
#include "linux.h"
int main(int argc,char * * argv)
```

```
{
hello_print("Hello");
linux_print("Linux");
}
/* hello.h */
#ifndef _HELLO_H
#define _HELLO_H
void hello_print(char * print_str);
#endif
/*  hello.c */
#include "hello.h"
void hello_print(char * print_str)
{
   printf("This is hello print %s\n",print_str);
}
/* linux.h*/
#ifndef _LINUX_H
#define _LINUX_H
void linux_print(char * print_str);
#endif
/* linux.c */
#include "linux.h"
void linux_print(char * print_str)
{
   printf("This is linux print %s\n",print_str);
}
```

交叉编译过程如下：

```
#arm-linux-gcc -c main.c
#arm-linux-gcc -c hello.c
#arm-linux-gcc -c linux.c
#arm-linux-gcc -o main main.o hello.o linux.o
```

输入如下命令可以查看生成的 main 文件的类型：

```
#file main
```

结果如下：

```
[root@localhost mydir]#file main
main: ELF 32-bit LSB executable, ARM, version 1, dynamically linked (uses shared libs), for
GNU/Linux 2.0.0, not stripped
```

从上面的结果可以看出，main 的文件类型为 ARM。到此交叉编译成功。若将其移植到 ARM 的目标机上，也可以正常运行。

小结

本章讲解了嵌入式 Linux 开发的知识，首先从 Linux 的基本知识、常用命令讲起，这是学习 Linux 的入门知识，然后通过实例讲述 Linux C 编程的基本过程及相应的开发工具，包

括 vi 和 emacs 编辑工具、gcc 编译工具、make 工程管理工具和 gdb 调试工具的使用方法，版本控制的基本概念，这些都是进行 Linux 开发必须掌握的工具；为了能够进行多进程和多线程的开发，还讲述了 Linux 下的进程和线程编程基本方法，相应的介绍了多进程和多线程的程序调试方法。最后讲解了交叉编译的概念，通过实例介绍了如何将一个 Linux 的程序交叉编译为在 ARM 处理器上运行的程序。

本章深入浅出，从 Linux 开发入门到较高级的应用编程，最后到交叉编译为不同处理器上运行的程序，这些都是学习嵌入式系统编程必不可少的知识，为今后学习嵌入式系统的开发奠定基础。

第7章

构建嵌入式 Linux 系统

学习目标

通过本章的学习，应该掌握：

- 构建嵌入式 Linux 系统的基本方法
- 嵌入式 Linux 交叉编译环境的建立方法
- Bootloader 的基本组成和移植
- Linux 系统的移植方法
- Linux 应用程序的下载和远程调试

嵌入式　　　系统的构建流程

问题：嵌入式 Linux 系统由哪些部分组成？应该按照什么样的步骤去构建一个嵌入式 Linux 系统？
重点：嵌入式 Linux 的构建流程。
内容：嵌入式 Linux 系统的组成和构建。

7.1.1　嵌入式 Linux 系统的组成

任务：了解嵌入式 Linux 系统的组成。

大多数嵌入式系统都使用 Flash 作为存储介质，并且要将 Flash 进行分区使用，嵌入式 Linux 在 Flash 上的存储由 3 个部分组成，如图 7-1 所示。

Bootloader及启动参数	内核映像	根文件系统

图 7-1　嵌入式 Linux 系统的组成

1．Bootloader 及启动参数

Bootloader 是系统的引导程序。在一般情况下，Bootloader 会被烧写到系统的启动地址处（对于 ARM 体系，一般为物理地址的 0x0）。当系统启动后，首先运行 Bootloader，在 Bootloader 的前面包含了系统的启动代码，它将完成系统硬件的初始化工作，之后进入 Bootloader 的环境。在 Bootloader 运行的情况下，用户可以根据它的功能选择进行相应的操作，Bootloader 通常都会提供下载、烧写等功能和简单的用户界面。Bootloader 最基本的功能是加载 Linux 内核并运行。启动参数存放如 IP 地址、串口波特率、要传递给内核的命令行参数等可设置的参数。

2．内核映像

Linux 内核映像实际上是经过编译生成的一段可执行程序。在完整的系统中，Linux 内核映像会被烧写到 Flash 的某段地址内，内核映像由 Bootloader 加载运行。在嵌入式 Linux 系统中，内核有可能被压缩到系统的内存中，也可以直接放置在可运行的地址处。Bootloader 加载内核可以有几种不同的模式，但是最终都是跳转到 Linux 内核的起始地址运行。内核运行时，可能需要从外部获取启动参数，因此 Bootloader 会将参数保存在参数区，在 Linux 内核启动的时候，将该参数传递给内核。

3．根文件系统

根文件系统是 Linux 内核启动后首先需要加载的文件系统，一般来说，根文件系统需要烧写到可以固化的存储器（如 Flash）中，由内核挂接。Linux 内核本身运行并不依赖于根文件系统，但是要实现一些基本的功能就需要根文件系统的支持。当根文件系统挂接完成后，内核可能会运行根文件系统中的程序。在需要交互的系统中会加载 shell 程序，这时将提供与桌面 Linux 一致的界面。但是，这些显然都不是 Linux 系统运行所必需的。

7.1.2 嵌入式 Linux 系统的构建

任务：了解嵌入式 Linux 系统的构建过程。

在嵌入式 Linux 系统的构建中，Bootloader 和 Linux 内核一般都有相对成熟的代码，因此主要工作有两步：第一步是根据系统硬件平台的状况进行移植；第二步是采取交叉编译的方法对源代码进行编译，形成运行时需要的映像(image)文件。

对于 Bootloader，用户需要针对本硬件平台完成移植。完成移植后，将移植部分和不需要改动的部分一起做交叉编译，生成可执行的二进制文件。不同 Bootloader 的功能不同，因此需要的硬件支持也不同，移植的部分也不一样。Bootloader 的移植结构如图 7-2 所示。

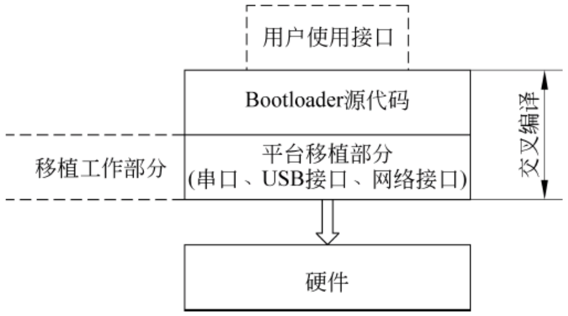


图 7-2 Bootloader 的移植结构

Linux 内核支持多种体系的处理器，在 Linux 的内核代码中，包括与体系结构无关和与体系结构相关的两个部分。在 Linux 的内核代码中有对各种体系结构(x86、ARM 等)提供支持的代码，但是针对某一具体系统的平台还需要完成对硬件的移植，主要工作包括系统配置(包括本系统的内存映射)、定时器及中断系统的移植(用于完成系统所需要的时钟)，串口驱动程序的移植(作为系统的标准输出，显示调试信息等)。移植完成后，选择需要的硬件平台及相关配置进行交叉编译，形成内核映像文件。Linux 内核的移植结构如图 7-3 所示。

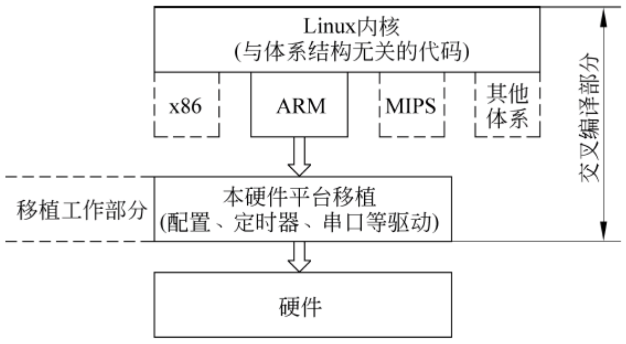


图 7-3 Linux 内核的移植结构

Linux 内核的启动并不依赖于根文件系统(rootfs)，但是要保证 Linux 内核正常运行，还需要根文件系统的支持。因此，在构建系统的过程中，需要为系统生成根文件系统的映像文件，让 Linux 内核在启动的时候挂接(mount)根文件系统。

宿主机和目标机之间的通信

问题：在使用开发主机-目标机开发模式时，怎样进行通信？
重点：开发主机-目标机的通信方式。
内容：Windows 超级终端、Linux 的 minicom、TFTP 协议和 NFS 网络共享。

7.2.1　宿主机和目标机

任务：理解宿主机和目标机之间的关系。

嵌入式系统内核和应用程序经过编译和链接后，可以下载到目标机，同时在程序运行过程中需要用户通过控制终端输入命令，并向用户显示特定信息，因此宿主机和目标机之间需要进行通信。

宿主机和目标机通信的物理通道一般有两种：串口和网口。其中串口是开发主机和目标机系统通信的基本手段，可以通过串口为目标机系统中的 Linux 建立一个控制终端，也可以完成内核和应用程序的下载。由于串口的速度比较慢，因此目前内核和应用程序的下载通常是通过网口使用 TFTP 工具完成的。在开发过程中，可以采用网络文件系统服务器 NFS，将开发主机的文件系统挂载到嵌入式系统中，可以在嵌入式系统控制终端上直接执行开发主机上的可执行程序。

本节将介绍 Windows 平台的超级终端、Linux 的 minicom、TFTP 协议和 NFS 网络共享 4 种通信方式。

7.2.2　Windows 的超级终端

任务：学会使用 Windows 超级终端。

超级终端是 Windows 自带的一个串口调试工具，其使用方法较为简单，被广泛应用在串口设备的初级调试上。超级终端是一个通用的串行交互软件，很多嵌入式应用系统都有与之交互的相应程序，通过这些程序，可以通过超级终端与嵌入式系统交互，使超级终端成为嵌入式系统的“显示器”。

超级终端的原理并不复杂，它会将用户输入随时发向串口(采用 TCP 协议时是发向网口的，这里只讲串口的情况)，但并不显示输入。它显示的是从串口接收到的字符。所以，嵌入式系统的相应程序应该完成如下任务。

(1) 将自己的启动信息、过程信息主动发到运行有超级终端的主机。

(2) 将从超级终端接收到的字符返回嵌入式系统进行处理，同时发送需要显示的字符(如命令的响应等)到主机。

在“开始”→“程序”→“附件”→“通信”菜单中选择“超级终端”选项，可以建立一个超级终端连接，界面如图 7-4 所示。

选择“文件”菜单中的“属性”选项，可以对其属性进行设置，要注意设置连接方式为 com

(串口),根据实际情况选择相应的串口,然后对其进行配置,配置对话框如图 7-5 所示。配置的参数要与目标处理器的 UART 端口的设置一致。

图 7-4 新建超级终端连接界面

图 7-5 串口属性设置

7.2.3 Linux 的 minicom

任务: 学会使用 Linux 的 minicom 软件。

串口上有很多通信软件,Linux 中应用最为广泛的是 minicom 软件。使用 minicom 可以在 Linux 下实现目标机和主机的连接。minicom 可以建立在 Linux 的串口设备(ttyS0)或者调制解调器(modem)设备上,它的操作界面没有 Windows 中的工具界面友好,需要从命令行启动。

打开一个终端,输入命令:minicom -s,打开 minicom 的配置界面,如图 7-6 所示。

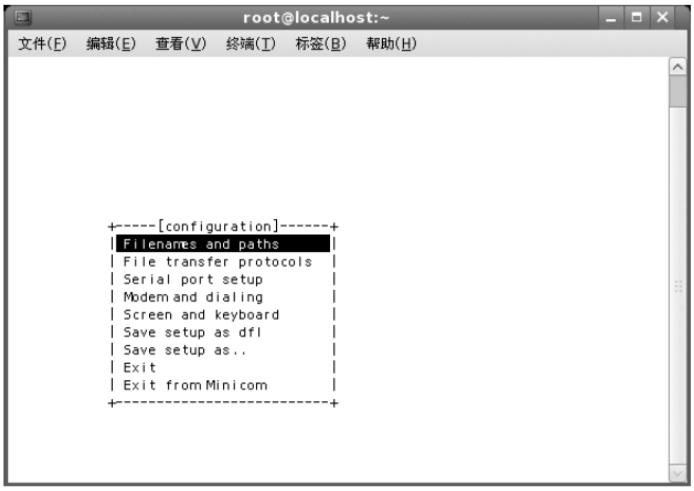


图 7-6 minicom 的配置界面

在主配置界面中,Exit from Minicom 表示退出 minicom 应用程序,Exit 表示退出配置

程序，进入 minicom 主程序。Save setup as dfl 和 Save setup as 表示保存当前的配置。当选择前者时，配置文件将保存在/etc/minirc. dfl 文件中。

配置的重要工作是配置串口，选择 Serial port setup 选项，配置界面如图 7-7 所示。在 minicom 串口配置界面中，可以使用 A～G 等字母选择相应的配置内容，A 用于选择串行设备，使用 Linux 下的设备名称，如/dev/modem、/dev/ttyS0 等。

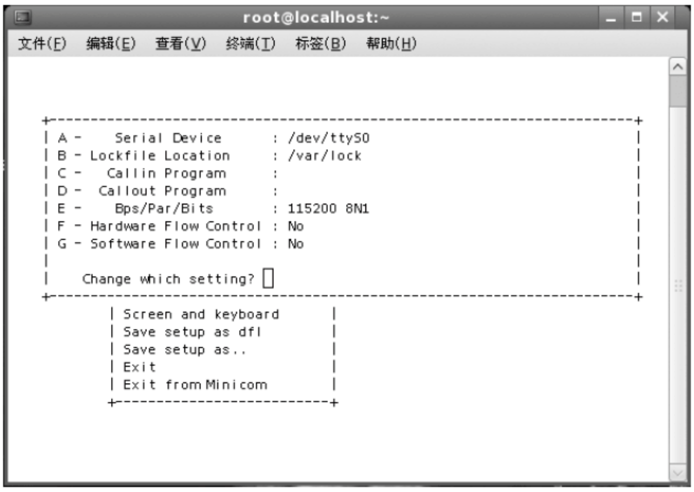


图 7-7　串口配置界面

在串口终端的配置中，需要设置的主要内容是波特率、数据位等内容，按 E 键启动波特率、数据位等的配置界面，如图 7-8 所示。串口波特率、数据位、奇偶校验位、停止位按 I 键实现(不同的开发板参数有所不同，应以参考手册为准)。按 F 键设置硬件流控制，一般选择 No。

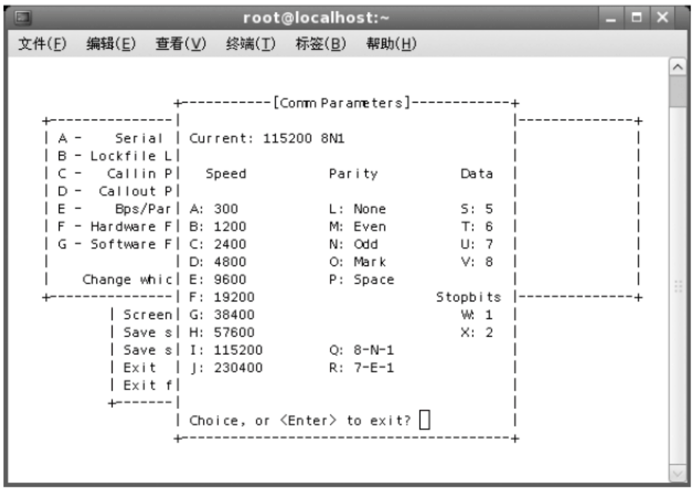


图 7-8　波特率、数据位等的配置界面

完成设置后按 Esc 键退出到 minicom 配置界面，保存设置，然后退出。

要想使 minicom 和目标机通信，只需输入命令 minicom 即可。比如在目标机上已经安装有 Linux 操作系统，则 minicom 启动后会自动启动目标机的操作系统，如图 7-9 所示。

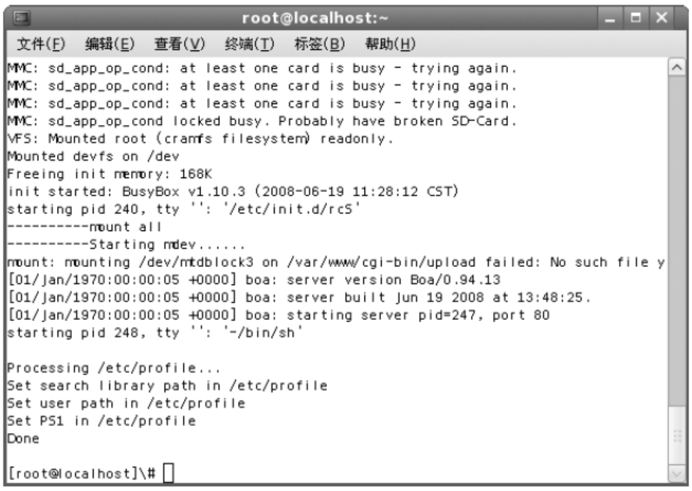


图 7-9 目标机的 Linux 系统界面

7.2.4 TFTP 协议

任务：学会安装和使用 TFTP 协议。

在宿主机和目标机系统之间通常使用网络协议实现文件传输，例如将主机端编译好的程序传输到目标机上。这时，一般需要在主机上开启网络协议的服务器端，而在目标机上也需要相应程序的支持(如某些 Bootloader 中的功能)。

TFTP(Trivial File Transfer Protocol，简单文件传输协议)的设计目的是传输小文件。TFTP 只能从文件服务器上获得或写入文件，不能列出目录，不进行认证，用于传输 8 位数据。

在使用 TFTP 工具前，需要首先确认是否安装了该协议，可以使用下面的命令进行确认：

```
[root@localhost ~]# rpm -qa|grep tftp
tftp-server-0.42-5
```

上面显示已经安装了 tftp-server-0.42-5 版本的协议。

若没有安装，可从相关的网站上下载此软件包，并使用下面的命令进行安装：

```
[root@localhost ~]# rpm -I tftp-server-0.42-5.i386.rpm
```

安装完成后还需要按以下方式进行设置。

第一步：创建文件 tftp，并将其保存在/etc/xinet.d 目录中，文件内容如下：

```
service tftp
{
        disable=no
        socket_type           =dgram
        protocol              =udp
        wait                  =yes
```

```
        user                    = root
        server                  = /usr/sbin/in.tftpd
        server_args             = -s /tftpboot
        per_source              = 11
        cps                     = 100 2
        flags                   = IPv4
}
```

第二步：启动 TFTP 服务器，需要重启 xinetd 服务，命令如下：

```
[root@ localhost xinetd.d]# service xinetd restart
```

显示信息如下：

```
停止 xinetd:                                          [确定]
启动 xinetd:                                          [确定]
```

7.2.5　NFS 网络共享

任务：学会配置和使用 Linux 下的 NFS。

NFS(Network File System，网络文件系统)是 FreeBSD 支持的文件系统中的一种，它允许一个系统在网络上与他人共享目录和文件。通过使用 NFS，用户和程序可以像访问本地文件一样访问远端系统上的文件。

在嵌入式系统中，常用的方式是由宿主机作为 NFS 服务器，目标机通过 NFS 挂载(mount)宿主机上的文件系统，这样在目标机的调试过程中，就可以使用宿主机上的文件，而不需要反复将程序烧写或者下载到目标机上。

要使用 NFS，首先要对其进行配置，但要注意，如果使用 Red Hat 等商业化的操作系统，需要关闭防火墙。

以 Fedora Core 8 为例，配置 NFS。首先在系统菜单中，选择“管理”→“服务器设置”→NFS 选项，打开“NFS 服务器配置方案”窗口，如图 7-10 所示。

在使用 NFS 共享一个文件或一个文件系统时，首先要设定该文件或文件系统所在的目录、主机的 IP 地址和对该文件或文件系统的读写权限，可以单击“服务器设置”按钮进行配置。另外，要新添加一个 NFS 共享目录，可单击“添加”按钮。图 7-11 所示为添加一个 NFS 目录。

图 7-10　“NFS 服务器配置方案”窗口

图 7-11　添加 NFS 共享目录

启动和停止 NFS 服务器,可以使用下面的命令:

```
[root@ localhost~ ]# /etc/rc.d/init.d/nfs start
Starting NFS services:                              [OK]
Starting NFS quotas:                                [OK]
Starting NFS daemon:                                [OK]
Starting NFS mountd:                                [OK]
[root@ localhost~ ]# /etc/rc.d/init.d/nfs stop
Shutting down NFS mountd:                           [OK]
Shutting down NFS daemon:                           [OK]
Shutting down NFS quotas:                           [OK]
Shutting down NFS services:                         [OK]
```

问题:Bootloader 的运行方式有哪些? Bootloader 具有什么样的功能?
重点:Bootloader 的功能。
内容:嵌入式 Linux 的引导方式,Bootloader 的功能。

在嵌入式系统中,由于其不具有自主开发的能力,除了要用 Bootloader(引导装载器)引导操作系统之外,还要利用其辅助开发,如实现与主机通信、与用户交互、更新系统等功能。Bootloder 是和目标系统的硬件架构密切相关的。如果需要一个针对目标机的立即可用的引导装载器,就必须进行一些移植的工作。一般会选择一个合适的 Bootloader,然后修改其与硬件相关的代码并进行配置来使用。

7.3.1 Bootloader 的作用

任务:了解 Bootloader 的作用。

Bootloader 是系统启动后首先运行的程序,对于嵌入式 Linux 操作系统的 Bootloader,其最基本的功能是加载 Linux 的内核并运行。在嵌入式系统开发完成后,Bootloader 的正常运行方式就是从指定的地址处自动运行 Linux 内核,将系统的控制权交给 Linux,然后由 Linux 实现系统所需要的各种功能。

实际上,各种 Bootloader 不仅局限于运行内核,还会有一些其他的功能。尤其是在开发的过程中,这些功能会十分有效。一般来说,Bootloader 的扩展功能包括以下几个部分。

(1) 通信功能

Bootloader 一般都具有和主机交互的功能,交互的媒介可能是串口、网口或者 USB 口。在很多 Bootloader 中,都会提供串口和网络接口的驱动,其中也包含简单协议栈。Bootloader 的通信功能一般需要根据本硬件平台进行一部分的移植,然后配合主机端的程序才可以使用。

具有通信功能以后,Bootloader 不但可以从主机下载 Linux 的内核和文件系统,而且可以构建用户与目标机的交互界面。

（2）Flash 的相关功能

在嵌入式系统的开发中，程序的固化是一个很重要的部分。尤其在形成产品之后，目标要脱离主机运行，程序必须处于固化状态。Flash 是嵌入式系统中固化程序的介质，Bootloader 对很多 Flash 介质都可以提供支持，基本功能是将映像文件（如 Linux 内核）烧写到 Flash 中。功能比较强大的 Bootloader 还支持对 Flash 的全面操作，甚至支持 Flash 文件系统。由于各种 Flash 介质具有相似的特点，Bootloader 对 Flash 提供的功能相似，但是具体到实际的系统，还需要做一定的移植和配置工作才可以使用。

（3）用户接口功能

用户接口功能是一种抽象的人机交互方式，它本身不依赖于传输的介质（例如串口、网络），Bootloader 的用户接口功能可让用户在主机端对 Bootloader 进行全面控制，从而实现对目标机的操作。简单的 Bootloader 可以提供菜单式交互模式，更为流行的交互模式是命令行式的，这样可以更方便地实现 Bootloader 的各种功能。

7.3.2　Bootloader 的启动方式

任务：理解 Bootloader 的两种启动方式和它们的应用场合。

CPU 通电后，会从某个地址开始执行。比如 ARM 结构的 CPU 会从地址 0x0000000 开始，需要把存储器件 ROM 或 Flash 等映射到这个地址，Bootloader 就存放在这个地址开始处，这样一通电就可以执行。

在开发时，通常需要使用各种命令操作 Bootloader，一般通过串口来连接 PC 和目标机，可以在串口上输入各种命令、观察运行结果等，这只对开发人员才有意义，用户使用产品时是不用接串口来控制 Bootloader 的。从这个角度来看，Bootloader 有两种操作模式：启动加载模式和下载模式，分别对应于嵌入式系统产品阶段和开发阶段。

1. 启动加载模式

启动加载（Bootloader）模式也称为自主模式（autonomous）。在这种情况下，Bootloader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行，此时，系统的控制权已经交给了操作系统，Bootloader 的任务完成。在整个过程中并没有用户的介入。这种模式是 Bootloader 的正常工作模式，因此在嵌入式产品发布的时候，Bootloader 显然是运行在这种模式下的。

2. 下载模式

在下载（Downloading）模式中，目标机的 Bootloader 将通过串口连接或网络连接等通信手段从主机下载文件，例如，下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 Bootloader 保存到目标机的 RAM 中，然后再被 Bootloader 写到目标机上的 Flash 类固态存储设备上。Bootloader 的这种模式通常在第一次安装内核与文件系统时使用。此外，以后的系统更新也会使用。工作于这种模式下的 Bootloader 通常都会向它的终端用户提供一个简单的命令行接口。

7.3.3　Bootloader 的两个阶段

任务：了解 Bootloader 的启动过程。

Bootloader 的启动过程可以分为单阶段(single stage)、多阶段(multi-stage)两种。通常多阶段的 Bootloader 能提供更为复杂的功能以及更好的可移植性。从固态存储设备上启动的 Bootloader 大多是两阶段的启动过程,分为 stage1 和 stage2 两大部分。依赖于 CPU 体系结构的代码,比如设备初始化代码等,通常都放在 stage1 中,而且通常都通过汇编语言来实现,以达到短小精悍的目的;stage2 则通常用 C 语言来实现,这样可以实现更复杂的功能,而且代码会具有更好的可读性和可移植性。

Bootloader 的 stage1 通常包括以下工作(按照执行的先后次序)。

(1) 硬件设备初始化。

(2) 为加载 Bootloader 的 stage2 准备 RAM 空间。

(3) 复制 Bootloader 的 stage2 到 RAM 空间中。

(4) 设置堆栈。

(5) 跳转到 stage2 的 C 入口点。

在 stage1 阶段进行的硬件初始化一般包括:关闭看门狗、关中断、设置 CPU 的速度和时钟频率、RAM 初始化等。这些并不都是必须的。比如在 S3C2410/S3C2440 的开发板所使用的 U-Boot 中,就将 CPU 的速度和时钟频率的设置放在了 stage2 中。甚至,将 stage2 的代码复制到 RAM 空间中也不是必须的,对于 Nor Flash 等存储设备,完全可以在上面直接执行代码,只不过和在 RAM 中执行相比效率低很多。

Bootloader 的 stage2 通常包括以下工作(按照执行的先后次序)。

(1) 初始化本阶段要用到的硬件设备。

(2) 检测系统内存映射(memory map)。

(3) 将内核映像和根文件系统映像从 Flash 设备上复制到 RAM 空间中。

(4) 设置内核启动参数。

(5) 调用启动内核。

为了方便开发,至少要初始化一个串口以便程序员与 Bootloader 进行交互。

所谓检测内存映射,就是确定板上使用了多少内存、它们的地址空间是什么。由于在嵌入式开发中 Bootloader 多是针对某类板子进行编写的,所以可以根据板子的情况直接设置,不需要考虑可以适用于各类情况的复杂算法。

将根文件系统映像复制到 RAM 中不是必须的,这取决于是什么类型的根文件系统,以及内核访问它的方法。

将内核存放到适当的位置后,直接跳到它的入口点即可调用内核。

7.3.4 常用 Bootloader 简介

任务: 了解常用的 Bootloader。

现在 Bootloader 种类繁多,比如 x86 上有 LILO、GRUB 等。对于 ARM 架构的 CPU,有 U-Boot、vivi 等。它们各有特点,下面列出 Linux 开放源码的 Bootloader 及其支持的体系架构,如表 7-1 所示。

表 7-1　开放源码的 Linux 引导程序

Bootloader	Monitor	描　　述	x86	ARM	PowerPC
LILO	否	Linux 磁盘引导程序	是	否	否
GRUB	否	GUN 的 LILO 替代程序	是	否	否
Loadlin	否	从 DOS 引导 Linux	是	否	否
ROLO	否	从 ROM 引导 Linux 而不需要 BIOS	是	否	否
Etherboot	否	通过以太网卡启动 Linux 系统的固件	是	否	否
LinuxBIOS	否	完全替代 BUIS 的 Linux 引导程序	是	否	否
BLOB	是	LART 等硬件平台的引导程序	否	是	否
U-Boot	是	通用引导程序	是	是	是
RedBoot	是	基于 eCos 的引导程序	是	是	是
vivi	是	Mizi 公司针对三星的 ARM CPU 设计的引导程序	否	是	否

对于 ARM S3C2410/S3C2440 来讲，一般选择 U-Boot 和 vivi。

vivi 是 Mizi 公司针对三星的 ARM 架构 CPU 专门设计的，基本上可以直接使用，命令简单方便。不过其初始版本只支持串口下载，速度较慢。在网上出现了各种改进版本，这些版本可以支持网络功能、USB 功能、烧写 YAFFS 文件系统映像等。

U-Boot 则支持许多 CPU，可以烧写 ext2、JFFS2 文件系统映像，支持串口下载、网络下载，并提供了大量的命令。相对于 vivi，它的使用比较复杂，但是可以用来更方便地调试程序。

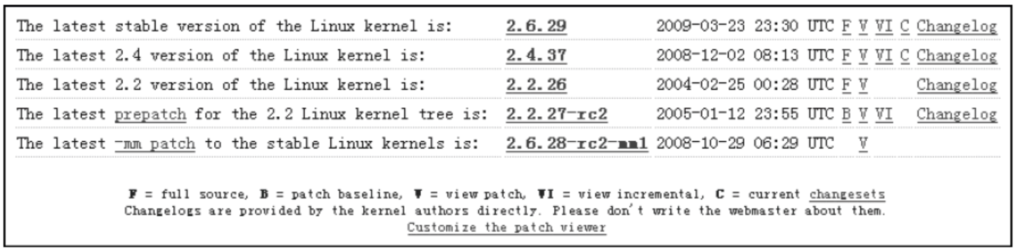
内核配置和移植

7.4.1　Linux 内核移植准备

任务： 了解 Linux 内核源码结构，学会分析 Makefile 文件和内核的 Kconfig 文件。

1. 获取内核源码

登录 Linux 内核的官方网站 http://www.kernel.org/，可以看到如图 7-12 所示的内容。



The latest stable version of the Linux kernel is:	2.6.29	2009-03-23 23:30 UTC	F V VI C	Changelog
The latest 2.4 version of the Linux kernel is:	2.4.37	2008-12-02 08:13 UTC	F V VI C	Changelog
The latest 2.2 version of the Linux kernel is:	2.2.26	2004-02-25 00:28 UTC	F V	Changelog
The latest prepatch for the 2.2 Linux kernel tree is:	2.2.27-rc2	2005-01-12 23:55 UTC	B V VI	Changelog
The latest -mm patch to the stable Linux kernels is:	2.6.28-rc2-mm1	2008-10-29 06:29 UTC	V	

F = full source, B = patch baseline, V = view patch, VI = view incremental, C = current changesets
Changelogs are provided by the kernel authors directly. Please don't write the webmaster about them.
Customize the patch viewer

图 7-12　kernel.org 网站首页

图 7-12 显示了 Linux 内核的最新稳定版本、正在开发的测试版本，图中间的版本号就是各种补丁的链接地址。图 7-12 中各种标记符的意义如表 7-2 所示。

表 7-2 kernel.org 网站首页各标记符的意义

标记符	描　　述
F	全部代码
B	当前的补丁基于哪个版本的内核，单击 B 链接可以下载这个内核
V	查看补丁文件的信息，哪些文件被修改了
VI	查看与上一个扩展版本相比，哪些文件被修改了
C	当前修改的记录，它的更新非常频繁，可以看到一天之内有几条更新记录
Changelog	这是正式的修改记录，由开发者提供

2. 内核源码结构

Linux 内核文件数目将近 2 万个，除去其他架构 CPU 的相关文件，支持 S3C2410、S3C2440 这两款芯片的完整内核文件有 1 万多个。这些文件分别位于顶层目录的 16 个子目录下，各个目录功能独立。表 7-3 描述了各个目录的功能，最后两个目录不包含内核代码。

表 7-3 Linux 内核子目录结构

目录名	描　　述
arch	包括所有与体系结构相关的内核代码。arch 的每一个子目录都代表 Linux 支持的一个体系结构。比如 arch/arm、arch/i386
crypto	常用加密和散列算法，还有一些压缩和 CRC 校验算法
drivers	存放系统所有的设备驱动程序，每种驱动程序都单独占一个子目录 比如 drivers/block：块设备驱动程序；driver/char：字符设备驱动程序。/driver/mtd 为 Nor Flash、Nand Flash 等设备的驱动程序
fs	Linux 所支持的文件系统代码，每一个子目录支持一个文件系统，如 JFFS2、ext2 等
include	包括编译内核所需要的头文件。与 ARM 相关的头文件在 include/asm-arm 子目录下
init	内核的初始化代码，但不是系统的引导代码，其中包含的 main.c 文件中的 start_kernel 函数是内核引导后运行的第一个函数
ipc	内核进程通信的代码
kernel	内核管理的核心代码，与处理器相关的代码位于 arch/ * /kernel/目录下
lib	内核用到的一些库函数代码，比如 string.c，与处理器相关的库函数代码位于 arch/ * /lib 目录下
mm	内存管理代码，与处理器相关的内存管理代码位于 arch/ * /mm/目录下
net	网络支持代码，每个子目录对应于网络的一个方面
security	安全、密钥相关的代码
sound	音频设备的驱动程序
usr	用来制作一个压缩的 cpio 归档文件：initrd 的镜像，它可以作为内核启动后挂接（mount）的第一个文件系统（一般用不到）
documentation	内核文档
scripts	用于配置、编译内核的脚本文件

3. Linux Makefile 分析

内核中的哪些文件将被编译？它们是怎样被编译的？它们连接时的顺序如何确定？哪些文件在最前面？哪些文件或函数先执行？这些都是通过 Makefile 来管理的。下面从最

简单的角度总结了 Makefile 的作用，有以下 3 点。

①决定编译哪些文件。

② 怎样编译这些文件。

③ 怎样链接这些文件，它们的顺序如何。

Linux 内核源码中含有很多个 Makefile 文件，这些 Makefile 文件又要包含其他一些文件（比如配置信息、通用规则等）。这些文件构成了 Linux 的 Makefile 体系，可以分为表 7-4 中的 5 类。

表 7-4 Linux 内核的 Makefile 文件分类

名　　称	描　　述
顶层 Makefile	它是所有 Makefile 文件的核心，从总体上控制着内核的编译、链接
.config	配置文件，在配置内核时生成。所有 Makefile 文件（包括顶层目录及各级子目录）都是根据.config 来决定使用哪些文件的
arch/ $(ARCH)/Makefile	对应体系结构的 Makefile，用来决定哪些与体系结构相关的文件参与内核的生成，并提供一些规则来生成特定格式的内核映像
scripts/Makefile.*	Makefile 共用的通用规则、脚本等
kbuild Makefile	各级子目录下的 Makefile，它们相对简单，被上一层 Makefile 调用来编译当前目录下的文件

内核文档 Documentation/kbuild/makefiles.txt 对内核中 Makefile 文件的作用、用法讲解得非常透彻，下面分析这 5 类文件。

（1）决定编译哪些文件

Linux 内核的编译过程从顶层 Makefile 开始，然后递归地进入各级子目录调用它们的 Makefile，分为以下 3 个步骤。

① 顶层 Makefile 决定内核根目录下的哪些子目录将被编译进内核。

② arch/ $(ARCH)/Makefile 决定 arch/ $(ARCH)目录下的哪些文件、哪些目录将被编译进内核。

③ 各级子目录下的 Makefile 决定所在目录下的哪些文件将被编译进内核，哪些文件将被编译成模块（即驱动程序），进入哪些子目录继续调用它们的 Makefile。

在步骤①，在顶层 Makefile 中可以看到如下内容：

```
434 init-y          :=init/
435 drivers-y       :=drivers/ sound/
436 net-y           :=net/
437 libs-y          :=lib/
438 core-y          :=usr/
…
564 core-y          +=kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

顶层的 Makefile 将目录分为 5 类：init-y、drivers-y、net-y、libs-y、core-y。表 7-3 中有 16 个子目录，除去 include 目录和后面的两个不包含内核代码的目录外，还有一个 arch 没有出现在内核中。它在 arch/ $(ARCH)/Makefile 中被包含进内核，在顶层 Makefile 中直接包含了这个 Makefile，如下所示：

```
include $(srctree)/arch/$(ARCH)/Makefile
```

对于 ARCH 变量，可以在执行 make 命令时传入，比如 make ARCH=arm…。另外，对于非 x86 平台，还需要指定交叉编译工具，这也可以在执行 make 命令时传入，比如 make CROSS_COMPILE=arm-linux…。为了方便，常在顶层 Makefile 中进行如下修改。

修改前：

```
ARCH            ?=$ (SUBARCH)
CROSS_COMPILE     ?=
```

修改后：

```
ARCH            ?=arm
CROSS_COMPILE     ?=arm-linux-
```

对于步骤②的 arch/$(ARCH)/Makefile，以 ARM 体系为例，在 arch/arm/Makefile 中可以看到如下内容：

```
94 head-y          :=arch/arm/kernel/head$ (MMUEXT).o arch/arm/kernel/init_task.o
…
173 core-y                    +=arch/arm/kernel/ arch/arm/mm/ arch/arm/common/
174 core-y                    +=$ (MACHINE)
175 core-$ (CONFIG_ARCH_S3C2410)      +=arch/arm/mach-s3c2400/
176 core-$ (CONFIG_ARCH_S3C2410)      +=arch/arm/mach-s3c2412/
177 core-$ (CONFIG_ARCH_S3C2410)      +=arch/arm/mach-s3c2440/

…
194 libs-y                    :=arch/arm/lib/ $ (libs-y)
…
```

从第 94 行可知，除前面的 5 类子目录外，又出现了另一类：head-y，不过它直接以文件名出现。MMUEXT 在 arch/arm/Makefile 的前面定义，对于没有 MMU 的处理器，MMUEXT 的值为-nommu，使用文件 head-nommu.S；对于有 MMU 的处理器，MMUEXT 的值为空，使用文件 head.S。

arch/arm/Makefile 中类似于第 173、174 行的代码进一步扩展了 core-y 的内容，第 194 行扩展了 libs-y 的内容，这些都是与体系结构相关的目录。175～177 行中的 CONFIG_ARCH_S3C2410 在配置内核时定义，它的值有 3 种：y、m 或空。y 表示编译进内核，m 表示编译为模块，空表示不使用。

编译内核时，将依次进入 init-y、core-y、libs-y、drivers-y 和 net-y 所列出的目录中执行它们的 Makefile，每个子目录都会生成一个 built-in.o（在 libs-y 所列出的目录中有可能生成 lib.a 文件）。最后，head-y 所表示的文件将和这些 built-in.o、lib.a 一起被连接成内核映像文件 vmlinux。

最后，关于第③步是怎么进行的介绍如下。

在配置内核时，生成配置文件.config（具体的配置过程将在 8.4.2 小节讲述）。内核顶层 Makefile 使用如下语句包含.config 文件，之后将根据.config 中定义的各个变量决定编译哪些文件。

```
-include .config.cmd

```

```
include .config
```

(2) 如何编译这些文件

即编译选项、链接选项是什么。这些选项分为 3 类：全局的，适用于整个内核代码树；局部的，仅适用于某个 Makefile 中的所有文件；个体的，仅适用于某个文件。

全局选项在顶层 Makefile 和 arch/$(ARCH)/Makefile 中定义，名称分别为：CFLAGS、AFLAGS、LDFLAGS、ARFLAGS，它们分别是编译 C 文件的选项、编译汇编文件的选项、链接文件的选项、制作库文件的选项。

局部选项在各个子目录中定义，名称分别为：EXTRA_CFLAGS、EXTRA_AFLAGS、EXTRA_LDFLAGS、EXTRA_ARFLAGS，它们的用途与全局选项相同，只是使用范围比较小，只针对当前 Makefile 中的所有文件。

另外，如果想针对某个文件定义它的编译选项，可以使用 CFLAGS_$@、AFLAGS_$@。前者用于编译某个 C 文件，后者用于编译某个汇编文件。$@表示某个目标文件名，比如以下代码表示编译 aha152x. c 时，选项中要额外加上-DAHA152X_STAT -DAUTOCONF。

```
#Makefile for linux/drivers/scsi
CFLAGS_aha152x.o =   -DAHA152X_STAT -DAUTOCONF
```

需要注意的是，这 3 类选项是一起使用的，在 scirpts/Makefile. lib 中可以看到：

```
_c_flags       =$(CFLAGS) $(EXTRA_CFLAGS) $(CFLAGS_$(*F).o)
_a_flags       =$(AFLAGS) $(EXTRA_AFLAGS) $(AFLAGS_$(*F).o)
```

(3) 如何链接这些文件，它们的顺序如何

前面分析有哪些文件要编译进内核时，顶层 Makefile 和 arch/$(ARCH)/Makefile 定义了 6 类目录(或文件)：head-y、init-y、drivers-y、net-y、libs-y 和 core-y。它们的初始值如下(以 ARM 体系为例)：

```
head-y        :=arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o
…
core-y                  +=arch/arm/kernel/ arch/arm/mm/ arch/arm/common/
core-y                  +=$(MACHINE)
core-$(CONFIG_ARCH_S3C2410)     +=arch/arm/mach-s3c2400/
core-$(CONFIG_ARCH_S3C2410)     +=arch/arm/mach-s3c2412/
core-$(CONFIG_ARCH_S3C2410)     +=arch/arm/mach-s3c2440/

…
libs-y                  :=arch/arm/lib/ $(libs-y)
…
```

在顶层 Makefile 中：

```
init-y         :=init/
drivers-y      :=drivers/ sound/
net-y          :=net/
libs-y         :=lib/
core-y         :=usr/
```

```
…
564 core-y          +=kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

可见,除 head-y 外,其余的 init-y、drivers-y、net-y、libs-y 和 core-y 都是目录名。在顶层 Makefile 中,这些目录名的后面直接加上 built-in. o 或 lib. a,表示要连接进内核的文件,如下所示:

```
575 init-y          :=$ (patsubst %/, %/built-in.o, $ (init-y))
576 core-y          :=$ (patsubst %/, %/built-in.o, $ (core-y))
577 drivers-y       :=$ (patsubst %/, %/built-in.o, $ (drivers-y))
578 net-y           :=$ (patsubst %/, %/built-in.o, $ (net-y))
579 libs-y1         :=$ (patsubst %/, %/lib.a, $ (libs-y))
580 libs-y2         :=$ (patsubst %/, %/built-in.o, $ (libs-y))
581 libs-y          :=$ (libs-y1) $ (libs-y2)
```

上面的 patsubst 是字符串处理函数,它的用法如下:

```
$ (patsubst pattern,replacement,text)
```

表示寻找 text 中符合格式 pattern 的字,用 replacement 替换它们。比如上面的 init-y 初值为 init/,经过第 575 行的交换后,init-y 变为 init/built-in. o。

顶层 Makefile 中后面的代码如下:

```
610 vmlinux-init :=$ (head-y) $ (init-y)
611 vmlinux-main :=$ (core-y) $ (libs-y) $ (drivers-y) $ (net-y)
612 vmlinux-all   :=$ (vmlinux-init) $ (vmlinux-main)
613 vmlinux-lds   :=arch/$ (ARCH) /kernel/vmlinux.lds
```

第 612 行的 vmlinux-all 表示所有构成内核映像文件 vmlinux 的目标文件,从第 610～612 行可知,这些目标文件的顺序为:head-y、init-y、core-y、libs-y、drivers-y、net-y,即 arch/arm/kernel/head. o、arch/arm/kernel/init_task. o、init/built. o、usr/built. o 等。

第 613 行表示链接脚本为 arch/ $ (ARCH) /kernel/vmlinux. lds。对于 ARM 体系,链接脚本就是 arch/arm/kernel/vmlinux. lds,它由 arch/arm/kernel/vmlinux. S 文件生成。

综上所述,Makefile 文件具有的功能可以总结如下。

① 配置文件. config 中定义了一系列的变量,Makefile 将结合它们来决定哪些文件被编译进内核、哪些文件被编译成模块、涉及哪些子目录。

② 顶层 Makefile 和 arch/ $ (ARCH) /Makefile 决定根目录下的哪些子目录、arch/ $ (ARCH) 目录下的哪些文件和目录将被编译进内核。

③ 各级子目录下的 Makefile 决定所在目录下的哪些文件将被编译进内核,哪些文件将被编译成模块,进入哪些子目录继续调用它们的 Makefile。

④ 顶层 Makefile 和 arch/ $ (ARCH) /Makefile 设置了可以影响所有文件的编译、链接选项:CFLAGS、AFLAGS、LDFLAGS、ARFLAGS;在各级子目录下的 Makefile 中可以设置能够影响当前目录下所有文件的编译、链接选项:EXTRA_CFLAGS、EXTRA_AFLAGS、EXTRA_LDFLAGS、EXTRA_ARFLAGS,还可以设置某个文件的编译选项 CFLAGS_ $ @、AFLAGS_ $ @。

⑤ 顶层 Makefile 按照一定的顺序组织文件,根据链接脚本 arch/arm/kernel/vmlinux.

lds 生成内核映像文件 vmlinux。

4. 内核的 Kconfig 分析

在内核目录下执行 make menuconfig 命令时，会看到如图 7-13 所示的界面，这就是内核的配置界面。

图 7-13 内核配置界面

内核源码的每个子目录中都有一个 Makefile 文件和 Kconfig 文件。Makefile 的作用前面已经讲过，Kconfig 用于配置内核，它就是各种配置界面的源文件。内核的配置工具能够读取各个 Kconfig 文件，生成配置界面供开发人员配置内核，最后生成配置文件.config。

内核的配置界面以树状的菜单形式组织，主菜单下有若干个子菜单，子菜单下又有子菜单或配置选项。每个子菜单或选项可以有依赖关系，用来确定它们是否显示。只有被依赖项的父项已经被选中，子项才会显示。

Kconfig 文件的语法可以参考 Documentation/kbuild/kconfig-language.txt 文件，下面讲述几个常用的语法，并在最后介绍菜单形式的配置界面操作方法。

Kconfig 文件的基本要素包括 config、menu、choice、comment、source 共 5 种条目。

(1) config 条目

config 条目常被其他条目包含，用来生成菜单、进行多项选择等。

config 条目用来配置一个选项，或者说，用来生成一个变量，这个变量会连同它的值一起被写入配置文件.config 中。比如有一个 config 条目用来配置 CONFIG_LEDS_S3C24XX，根据用户的选择，.config 文件中可能出现下面 3 种配置结果中的一个。

```
CONFIG_LEDS_S3C24XX=y      #对应的文件被编译进内核
CONFIG_LEDS_S3C24XX=m      #对应的文件被编译成模块
CONFIG_LEDS_S3C24XX        #对应的文件没有被使用
```

比如选择 fs/Kcofig 文件中的一段代码，配置 CONFIG_EXT3_FS_POSIX_ACL 选项：

```
115 config EXT3_FS_POSIX_ACL
```

```
116         bool "Ext3 POSIX Access Control Lists"
117         depends on EXT3_FS_XATTR
118         select FS_POSIX_ACL
119         help
120           Posix Access Control Lists (ACLs) support permissions for users and
121           groups beyond the owner/group/world scheme.
122
123           To learn more about Access Control Lists, visit the Posix ACLs for
124           Linux website <http://acl.bestbits.at/>.
125
126           If you don't know what Access Control Lists are, say N
127
128     config EXT3_FS_SECURITY
129         bool "Ext3 Security Labels"
```

代码中几乎包含了所有的元素，下面一一说明。

在第 115 行中，config 是关键字，标识一个配置选项开始；紧跟着的 EXT3_FS_POSIX_ACL 是配置选项的名称，省略了前缀 CONFIG_。

在第 116 行中，bool 表示变量类型，即 EXT3_FS_POSIX_ACL 的类型。类型有 5 种：bool、tristate、string、hex 和 int，其中的 tristate 和 string 是基本类型，其他类型是它们的变种。bool 有两种取值：y 和 n；tristate 变量取值有 3 种：y、n 和 m；string 变量取值为字符串；hex 变量取值为十六进制的数据；int 变量取值为十进制的数据。

bool 之后的字符串是提示信息，在配置界面中上下移动光标选中它时，就可以通过按空格键或 Enter 键来设置 EXT3_FS_POSIX_ACL 的值。提示信息的完整格式如下：

```
"prompt" <prompt> ["if" <expr>]
```

如果使用 if<expr>，则当 expr 为真时才显示提示信息。在实际使用时，prompt 关键字可以省略。

第 117 行表示依赖关系，格式如下：

```
"depends on"/"requires" <expr>
```

只有 Ext3 POSIX Access Control Lists 配置选项被选中时，当前配置选项的提示信息才会出现，才能设置当前配置选项。注意，如果依赖条件不满足，则取默认值。若要设置默认值，可以在其下一行添加 default y，表示默认选中。设置默认值的格式如下：

```
"default" <expr> ["if" <expr>]
```

第 118 行表示当前配置选项 CONFIG_EXT3_FS_POSIX_ACL 被选中时，配置选项 FS_POSIX_ACL 也会被自动选中，格式如下：

```
"select" <symbol> ["if" <expr>]
```

第 119 行表示下面几行是帮助信息，帮助信息的关键字有两种，如下：

```
"help" or "---help---"
```

它们完全一样。当遇到一行的缩进距离小时，表示帮助信息已经结束。比如第 127 行

为一空行，128 行的缩进距离比第 126 行的缩进距离小，帮助信息到第 126 行结束，第 128 行又开始新的 config 条目。

（2）menu 条目

menu 条目用于生成菜单，格式如下：

```
"menu" <prompt>
<menu option>
<menu block>
"endmenu"
```

它的实际使用并不如它的标准格式那样复杂，例如：

```
menu "Floating point emulation"
config FPE_NWFPE
        ...
config FPE_NWFPE
        ...
endmenu
```

menu 之后的字符串是菜单名，menu 和 endmenu 之间有很多 config 条目。在配置界面上会出现如下所示的菜单，移动光标选中它后按 Enter 键进入，就会看到这些 config 条目定义的配置选项。

```
Floating point emulation --->
```

（3）choice 条目

choice 条目将多个类似的配置选项组合在一起，供用户单选或多选，格式如下：

```
"choice"
<choice options>
<choice block>
"endchoice"
```

在实际使用中，也是在 choice 和 endchoice 之间定义多个 config 条目，比如 arch/arm/Kconfig 中有如下代码：

```
choice
    prompt "ARM system type"
    default ARCH_VERSATILE

config ARCH_AAEC2000
    ...
config ARCH_INTEGRATOR
    ...
endchoice
```

prompt "ARM system type"给出提示信息 ARM system type，用光标选中后按 Enter 键进入，就可以看到多个 config 条目定义的配置选项。

在 choice 条目中定义的变量类型只能有两种：bool 和 tristate，不能同时有这两种类型的变量。对于 bool 类型的 choice 条目，只能在多个选项中选择一个；对于 tristate 类型的

choice 条目,要么把多个(可以是一个)选项都设为 m;要么像 bool 类型的 choice 条目一样,只能选择一个。这是可以理解的,比如对于同一个硬件,它有多个驱动程序,可以选择将其中之一编译进内核中(配置选项为 y),或者把它们都编译为模块(配置选项设为 m)。

(4) comment 条目

comment 条目用于定义一些帮助信息,放在配置界面的第一行,并且这些帮助信息会出现在配置文件中(作为注释),格式如下:

```
"comment" <prompt>
<comment options>
```

在实际使用中也很简单,比如 arch/arm/Kconfig 中有如下代码:

```
menu "Floating point emulation"

comment "At least one emulation must be selected"
…
```

显示菜单 Floating point emulation--->之后,在第一行会看到如下内容:

```
---At least one emulation must be selected
```

而在.config 文件中也会看到如下内容:

```
#
#At least one emulation must be selected
#
```

(5) source 条目

source 条目用于读入另一个 Kconfig 文件,格式如下:

```
"source" <prompt>
```

下面是一个例子,取自 arch/arm/Kconfig 文件,表示读入 net/Kconfig 文件:

```
source "/net/Kconfig"
```

(6) 菜单形式的配置界面的操作方法

配置界面的开始几行就是它的操作方法,如图 7-14 所示。

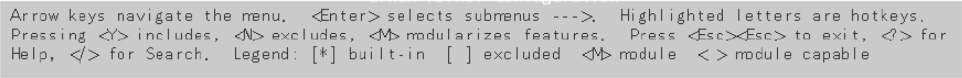


图 7-14 菜单形式的配置界面的操作方法

内核 scripts/kconfig/mconf.c 文件的注释给出了更详细的操作方法,讲解如下。

一些特定功能的文件可以直接编译进内核中,或者编译成一个可加载模块,或者根本不使用它们。还有一些内核,必须给它们赋一个值:十进制数、十六进制数,或者一个字符串。

在配置界面中,以[*]、＜M＞或[]开头的选项分别表示相应功能的文件被编译进内核中、被编译成一个模块、没有使用。＜＞表示相应功能的文件可以被编译成模块。

要修改配置选项,先使用方向键高亮选中它,按 Y 键选择将它编译进内核,按 M 键选

择将它编译成模块,按 N 键将不使用它。也可以按空格键进行循环选择,例如:Y->N->M->Y。

上/下方向键用来高亮选中某个配置选项,如果要进入某个子菜单,先选中它,然后按 Enter 键进入。配置选项的名称后有--->表示它是一个子菜单。配置选项的名称中有一个高亮的字母,称为“热键”(hotkey),直接输入热键可以选中该配置选项,或者循环选中具有相同热键的配置选项。

可以使用翻页键 PgUp 和 PgDn 移动配置界面中的内容。

要退出配置界面,使用左/右方向键选择<Exit>选项,然后按 Enter 键。如果没有配置选项使用后面这些按键作为热键,也可以按两次 Esc 键或 E 或 X 键退出。

按 Tab 键可以在<Select>、<Exit>和<Help>这 3 个选项间进行循环选择。

要想阅读某个配置选项的帮助信息,选中它之后,再选择<Help>选项,按 Enter 键;也可以选中配置选项后,直接按 H 或? 键。

对于 choice 条目中的多个配置选项,使用方向键高亮选中某个配置选项,按 S 键或空格键选中它们;也可以通过输入配置选项的首字母,然后按 S 键或空格键选中它。

对于 int、hex 或 string 类型的配置选项,要输入它们的值时,先高亮选中它,按 Enter 键,输入数据,再按 Enter 键。对于十六进制数据,前缀 0x 可以省略。

配置界面的最下面有如下两行:

```
Load an Alternate Configuration File
Save an Alternate Configuration File
```

前者用于加载某个配置文件,后者用于将当前的配置保存到某个配置文件中去。需要注意的是,如果不使用这两个选项,配置的加载文件、输出文件都默认为. config 文件;如果加载了其他的文件(假设文件名为 A),然后在它的基础上进行修改,最后退出保存时,这些变动会保存到 A 中去,而不是. config 中。

当然,可以先加载(Load an Alternate Configuration File)文件 A,然后修改,最后保存(Save an Alternate Configuration File)到. config 中去。

7.4.2 内核的配置

任务:掌握各种内核配置命令的使用方法,理解内核配置选项的含义。

配置 Linux 内核是创建一个能运行在目标系统上的内核的第一步。配置内核有很多种方法,在配置内核的过程中还有很多选项需要选择。不论使用哪种方法或选择哪些选项配置内核,内核都会在配置完成后生成一个. config 文件,用来保存当前配置的所有选项的设置信息。

1. 内核配置命令

Linux 提供了多种内核配置命令,各种命令的用法如下。

(1) make config

提供一个命令行界面,然后对每一个内核选项依次询问用户的选择,如下例所示:

```
[root@ localhost kernel-2.6.13]#make config
```

```
HOSTCC   scripts/basic/fixdep
HOSTCC   scripts/basic/split-include
  HOSTCC   scripts/basic/docproc
  HOSTCC   scripts/kconfig/conf.o
  HOSTCC   scripts/kconfig/kxgettext.o
  HOSTCC   scripts/kconfig/mconf.o
  HOSTCC   scripts/kconfig/zconf.tab.o
  HOSTLD   scripts/kconfig/conf
scripts/kconfig/conf arch/arm/Kconfig
#
# using defaults found in .config
#
*
* Linux Kernel Configuration
*
*
* Code maturity level options
*
Prompt for development and/or incomplete code/drivers (EXPERIMENTAL) [Y/n/?]
```

（2）make menuconfig

提供一个文本图形界面的配置菜单，其中列出了内核所能提供的全部功能，可以根据需要选择配置选项，该方法使用起来比较简单，一般使用这种方法进行内核配置。配置界面如图 7-15 所示。

图 7-15　make menuconfig 配置界面

进行配置时有 3 种选择，它们代表的含义分别说明如下。

Y：将该功能编译进内核

N：不将该功能编译进内核

M：将该功能编译成可以在需要时动态插入到内核中的模块

（3）make xconfig

提供一个基于 GTK 的 X-Window 图形界面配置菜单，界面最为友好，使用鼠标就可以选择对应的选项。

(4) make oldconfig

和 make config 类似，不过它只提示用户配置不正确的选项。

2. 内核配置选项

内核的配置选项很多，需要根据目标机的实际情况进行选择。然而，并不是所有的选项都需要设置。一般的做法是在某个默认配置文件的基础上进行修改，比如可以先加载配置文件、arch/arm/configs/s3c2410_defconfig，再增加、去除某些配置选项。

由于某些选项对嵌入式 Linux 系统意义不大，下面只介绍重要的内核选项，如表 7-5 所示。

表 7-5　配置界面主菜单的类别和功能

配置界面主菜单	描　　述
Code maturity level options	代码成熟度选项：用于包含一些正在开发的或者不成熟的代码、驱动程序。一般不设置
General setup	常规设置：比如增加附加版本号、支持内存页交换(swap)功能、System V 进程间通信等。一般使用默认配置
Loadable module support	可加载模块支持：一般会打开可加载模块支持(enable loadable module support)、允许卸载已经加载的模块(module unloading)、让内核通过运行 modprobe 来自动加载所需要的模块(automatic kernel module loading)
System Type	系统类型：选择 CPU 的架构、开发板类型等与开发板相关的配置选项
Kernel Features	用于设置内核的一些参数，比如是否支持内核抢占(这对实时性有帮助)、是否支持动态修改系统时钟(timer tick)等
Networking	网络协议选项：一般都选择 Networking support 选项以支持网络功能，选择 Packet socket 以支持 socket 接口功能，选择 TCP/IP networking 选项以支持 TCP/IP 网络协议。通常可以在选择 Networking support 选项后，使用默认配置
Device Drivers	设备驱动程序：几乎包含了 Linux 的所有驱动程序
File systems	文件系统：可以在其中选择要支持的文件系统，比如 ext2、JFFS2 等

7.4.3　Linux 内核的编译

任务： 掌握 Linux 内核编译命令，掌握内核编译步骤。

如果已多次编译过内核，为了把内核源码恢复到“干净”的初始状态，在编译之前通常需要清除一些文件，使用的命令如下。

(1) make clean

清除以前编译内核时产生的所有目标文件、模块文件、核心以及一些临时文件等，不产生任何文件。

(2) make rmproper

删除以前在构核过程中所产生的中间文件，除了执行 make clean 命令外，还要删除 .config、.depend 等文件，把核心源码恢复到最原始的状态。下次编译内核时必须进行重新配置。

编译内核通常分为以下几个步骤。

(1) make dep(建立依赖关系，在 Linux-2.6.x 中，这一步已经不再需要)

内核中的 C 源程序文件和头文件之间存在一定的依赖关系，内核的 Makefile 必须知道

这些依赖关系才能确定需要编译哪些源文件。但是，使用 make menuconfig 命令配置完内核之后，该命令会自动生成一些编译所需要的头文件。因此，当改变内核配置之后，在编译之前应该重新建立正确的依赖关系。

（2）make zImage

zImage 目标的含义是编译一个使用 gzip 算法压缩的内核映像 zImage；另外其他的内核映像还有 bzImage 和 vmlinux，bzImage 的含义是 big zImage，该映像使用 gzip 算法，但确保生成的压缩率较高的内核可以工作；vmlinux 目标的含义是只编译一个非压缩的内核映像 vmlinux。make zImage 命令执行完成后，既生成 zImage，也生成 vmlinux。

如果编译成功，所生成的内核映像文件将放置在 arch/${ARCH}/boot 目录下，对于 ARM 架构来说就是 arch/arm/boot。

（3）make modules

如果在配置内核时选择了对内核模块的支持，则使用该命令单独编译内核模块。

（4）make modules_install

如果需要安装内核模块，则使用该命令将模块文件复制到/lib/modules/＜新内核版本号＞/下，并运行 depmod 生成 modules.dep 内核模块的依赖关系配置文件。insmod、modprobe 需要用它去加载内核所需的驱动程序。

构建嵌入式根文件系统

根文件系统是嵌入式 Linux 的三个基本部分之一，本节将首先介绍 Linux 中文件系统的概念和各种文件系统，然后介绍 Linux 中根文件系统的结构。

7.5.1 Linux 下的文件系统

任务：掌握什么是文件系统，什么是根文件系统。

文件系统（File System）是文件存放在磁盘等存储设备上的组织方法，主要作用体现在对文件和目录的组织上。目录提供了管理文件的一个方便而有效的途径。

类似于 Windows 下的 C、D、E 等各个盘，Linux 系统也可以将磁盘、Flash 等存储设备划分为若干个分区，在不同分区存放不同类别的文件。与 Windows 的 C 盘类似，Linux 也要在一个分区上存放系统启动所必需的文件，比如内核映像文件（在嵌入式系统中，内核一般要单独存放在一个分区中）、内核启动后运行的第一个程序（init）、给用户提供操作界面的 shell 程序、应用程序所依赖的库等。这些必需的、基本的文件组成了根文件系统，它们存放在一个分区中。Linux 系统启动后首先挂接这个分区，称为挂接（mount）根文件系统。其他分区上的所有目录、文件的集合也称为文件系统。

在一个分区上存储文件时，需要遵循一定的格式，这种格式称为文件系统类型，比如 FAT16、FAT32、NTFS、ext2、ext3、JFFS2、yaffs 等。另外，Linux 还有几种虚拟的文件系统类型，比如 proc、sysfs 等，它们的文件并不存储在实际的设备上，而是在访问它们时由内核临时生成的。比如 proc 文件系统中的 uptime 文件，读取它时可以得到两个时间值（用来表示系统启动后运行的秒数、空闲的秒数），每次读取时都有内核即刻生成，每次读取结果都不

一样。

7.5.2 嵌入式 Linux 的文件系统

任务：了解常见的文件系统有哪些，各有什么特点。

Linux 启动时，第一个必须挂载的是根文件系统；若系统不能从指定设备上挂载根文件系统，则系统会出错而退出启动。之后可以自动或手动挂载其他的文件系统，因此，一个系统中可以同时存在不同的文件系统。

不同的文件系统类型有不同的特点，因而根据存储设备的硬件特性、系统需求等有不同的应用场合。在嵌入式 Linux 应用中，主要的存储设备为 RAM（DRAM，SDRAM）和 ROM（常采用 Flash 存储器），常用的基于存储设备的文件系统类型包括 JFFS2、yaffs、cramfs、romfs、ramdisk、ramfs/tmpfs 等。

1. 基于 Flash 的文件系统

Flash（闪存）作为嵌入式系统的主要存储媒介，有其自身的特性。因此，必须针对 Flash 的硬件特性设计符合应用要求的文件系统，传统的文件系统如 ext2 等，用做 Flash 的文件系统会有诸多弊端。

在嵌入式 Linux 下，MTD（Memory Technology Device，存储技术设备）为底层硬件（闪存）和上层（文件系统）之间提供一个统一的抽象接口，即 Flash 的文件系统都是基于 MTD 驱动层的。使用 MTD 驱动程序的主要优点在于，它是专门针对各种非易失性存储器（以闪存为主）而设计的，它提供了基于扇区的擦除和读写操作的更好的接口，因而对 Flash 具有更好的支持。

一块 Flash 芯片可以被划分为多个分区，各分区可以采用不同的文件系统；两块 Flash 芯片也可以合并为一个分区使用，采用一个文件系统，即文件系统是针对存储器分区而言的，而非存储芯片。

（1）JFFS2（Journalling Flash File System v2）

JFFS 文件系统最早是由瑞典 Axis Communications 公司基于 Linux 2.0 的内核为嵌入式系统开发的文件系统。JFFS2 是 RedHat 公司基于 JFFS 开发的闪存文件系统，最初是针对 RedHat 公司的嵌入式产品 eCos 开发的嵌入式文件系统，所以 JFFS2 也可以用在 Linux、uCLinux 中。

JFFS 文件系统主要用于 NOR 型闪存，基于 MTD 驱动层，特点是：可读写的、支持数据压缩的、基于哈希表的日志型文件系统，并提供了崩溃/掉电安全保护，提供“写平衡”支持等。缺点主要是当文件系统已满或接近满时，因为垃圾收集的关系而使 JFFS2 的运行速度大大降低。

JFFS 不适合用于 NAND 闪存主要是因为 NAND 闪存的容量一般较大，这样导致 JFFS 为维护日志节点所占用的内存空间迅速增大，另外，JFFSx 文件系统在挂载时需要扫描整个 Flash 的内容，以找出所有的日志节点，建立文件结构，对于大容量的 NAND 闪存会耗费大量时间。

（2）yaffs（Yet Another Flash File System）

yaffs/yaffs2 是专为嵌入式系统使用 NAND 型闪存而设计的一种日志型文件系统。与

JFFS2 相比，它减少了一些功能(例如不支持数据压缩)，所以速度更快，挂载时间更短，对内存的占用较小。另外，它还是跨平台的文件系统，除了 Linux 和 eCos，还支持 Windows CE、pSOS 和 ThreadX 等。

yaffs/yaffs2 自带 NAND 芯片的驱动程序，并且为嵌入式系统提供了直接访问文件系统的 API，用户可以不使用 Linux 中的 MTD 与 VFS，直接对文件系统进行操作。当然，yaffs 也可与 MTD 驱动程序配合使用。

yaffs 与 yaffs2 的主要区别在于，前者仅支持小页(512B) NAND 闪存，后者则可支持大页(2KB) NAND 闪存。同时，yaffs2 在内存空间占用、垃圾回收速度、读/写速度等方面均有大幅提升。

(3) cramfs(Compressed ROM File System)

cramfs 是 Linux 的创始人 Linus Torvalds 参与开发的一种只读的压缩文件系统。它也基于 MTD 驱动程序。

在 cramfs 文件系统中，每一页(4KB)被单独压缩，可以随机访问，其压缩比高达 2∶1，为嵌入式系统节省大量的 Flash 存储空间，使系统可通过更低容量的 Flash 存储相同的文件，从而降低系统成本。

cramfs 文件系统以压缩方式存储，在运行时解压缩，所有的应用程序都要求复制到 RAM 中运行，由于 cramfs 是采用分页压缩的方式存放文件的，在读取文件时，不会很快就耗用过多的内存空间，只针对目前实际读取的部分分配内存，尚没有读取的部分不分配内存空间，当读取的文件不在内存中时，cramfs 文件系统自动计算压缩后的文件存放的位置，再即时解压缩到 RAM 中。

另外，它的速度快，效率高，其只读的特点有利于保护文件系统免受破坏，提高了系统的可靠性。由于具有以上特性，cramfs 在嵌入式系统中应用广泛，但是它的只读属性同时又是它的一大缺陷，使得用户无法对其内容进行扩充。

cramfs 映像通常存放在 Flash 中，但是也能存放在别的文件系统里，使用 loopback 设备可以把它安装到别的文件系统里。

(4) romfs

传统型的 romfs 文件系统是一种简单的、紧凑的、只读的文件系统，不支持动态擦写保存，按顺序存放数据，因而支持应用程序以 XIP(eXecute In Place，片内运行)方式运行，在系统运行时，节省 RAM 空间。uClinux 系统通常采用 romfs 文件系统。

其他文件系统：FAT/FAT32 也可用于实际嵌入式系统的扩展存储器(例如 PDA、Smartphone、数码相机等的 SD 卡)，这主要是为了更好地与最流行的 Windows 桌面操作系统相兼容。ext2 也可以作为嵌入式 Linux 的文件系统，不过将它用于 Flash 闪存会有诸多弊端。

2. 基于 RAM 的文件系统

(1) ramdisk

ramdisk 是将一部分固定大小的内存当做分区来使用。它并非一个实际的文件系统，而是一种将实际的文件系统装入内存的机制，并且可以作为根文件系统。将一些经常被访问而又不会更改的文件(如只读的根文件系统)通过 ramdisk 放在内存中，可以明显地提高系统的性能。

在 Linux 的启动阶段，initrd 提供了一套机制，可以将内核映像和根文件系统一起载入内存。

（2） ramfs/tmpfs

ramfs 是 Linus Torvalds 开发的一种基于内存的文件系统，工作于虚拟文件系统（VFS）层，不能格式化，可以创建多个，在创建时可以指定其最大能使用的内存空间大小。

ramfs/tmpfs 文件系统把所有的文件都放在 RAM 中，所以读/写操作发生在 RAM 中，可以用 ramfs/tmpfs 来存储一些临时性或经常要修改的数据，例如/tmp 和/var 目录，这样既避免了对 Flash 存储器的读/写损耗，也提高了数据的读/写速度。

ramfs/tmpfs 相对于传统的 ramdisk 的不同之处主要在于：不能格式化，文件系统大小可随所含文件内容大小变化。

7.5.3　Linux 根文件系统目录结构

任务：掌握 Linux 根文件系统的目录结构和各个目录的作用。

Linux 根文件系统中一般有如图 7-16 所示的目录结构。下面将依次介绍这些目录的作用。

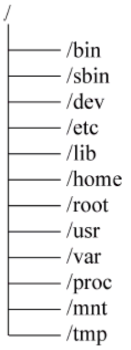


图 7-16　Linux 根文件系统的目录结构

1. /bin 目录

该目录下存放所有用户（包括系统管理员和一般用户）都可以使用的基本命令，这些命令在挂接其他文件系统之前就可以使用，所以/bin 目录必须和根文件系统在同一个分区中。

/bin 目录下常用的命令有 cat、chmod、cp、ls 等。

2. /sbin 目录

该目录下存放系统命令，即只有管理员能够使用的命令，系统命令还可以存放在/usr/sbin、/usr/local/sbin 目录下。/sbin 目录中存放的是系统的基本命令，用于启动系统、修复系统等。与/bin 目录类似，在挂接其他文件系统之前就可以使用/sbin，所以/sbin 必须和根文件系统在同一个分区中。

/sbin 下常用的命令有 shotdown、reboot、fdisk 等。

不是急迫需要使用的命令存放在/usr/sbin 目录下。本地安装的系统命令存放在/usr/local/sbin 目录下。

3. /dev 目录

该目录下存放的是设备文件。设备文件是 Linux 中特有的文件类型，在 Linux 系统中，以文件的方式访问各种外围设备，即通过读/写某个设备文件操作某个具体硬件。比如通过/dev/ttySAC0 可以访问串口 0，通过/dev/mtdblock1 可以访问 MTD 设备（NAND Flash、NOR Flash 等）的第 2 个分区。

设备文件有两种：字符设备和块设备。在 PC 上执行命令：ls /dev/ttySAC0 /dev/hda1 -l，可以看到如下结果：

```
[root@ localhost ~]# ls /dev/ttySAC0 /dev/hda1 -l
brw-r--r--1 root root 3,   1 03-30 22:59 /dev/hda1
```

```
crw-r--r--1 root root 4, 64 03-30 22:58 /dev/ttySAC0
```

其中,字母 b 表示这是一个块设备文件,c 表示这是一个字符设备文件;"3,1"、"4,64"等表示的是设备的主次设备号。

设备文件可以使用 mknod 命令创建,比如:

```
mknod /dev/ttySAC0 c 4 64
mknod /dev/hda1 b 3 1
```

4. /etc 目录

如表 7-6、表 7-7 所示,该目录下存放的是各种配置文件。对于 PC 上的 Linux 系统,/etc 目录下的子目录和文件非常多。这些目录和文件都是可选的,它们依赖于系统中具有的应用程序,依赖于这些程序是否需要配置文件。在嵌入式系统中,这些内容可以大为精简。

表 7-6 /etc 下的子目录

目录	描　述	目录	描　述
opt	用来配置/opt 下的程序(可选)	sgml	用来配置 SGML(可选)
X11	用来配置 X Window(可选)	xml	用来配置 XML 下的程序(可选)

表 7-7 /etc 下的文件

文　件	描　述
export	用来配置 NFS 文件系统(可选)
fstab	用来指明当执行 mount -a 命令时需要挂接的文件系统(可选)
mtab	用来显示已经加载的文件系统(可选)
ftpuuers	启动 FTP 服务时,用来配置用户访问权限(可选)
group	用户的组文件(可选)
inittab	Init 进程的配置文件(可选)
ld. so. conf	其他共享库的路径(可选)
passwd	密码文件(可选)

5. /lib 目录

该目录下存放共享库和可加载模块(驱动程序),共享库用于启动系统、运行根文件系统中的可执行程序,比如/bin、/sbin 目录下的程序。其他不是根文件系统所必需的库文件可以存放在其他目录下,比如/usr/lib、/usr/X11R6/lib、/var/lib 等。

表 7-8 所示是/lib 目录中的内容。

表 7-8 /lib 目录中的内容

目录/文件	描　述	目录/文件	描　述
libc. so. *	动态链接 C 库(可选)	modules	内核可加载模块存放的目录(可选)
ld *	链接器、加载器(可选)		

6. /home 目录

用户目录,是可选的。对于每个普通用户,在/home 目录下都有一个以用户名命名的子目录,里面存放与用户相关的配置文件。

7. /root 目录

根用户(用户名为 root)的目录,与此对应,普通户的目录是/home 下的某个子目录。

8. /usr 目录

/usr 目录的内容可以存放在另一个分区中,在系统启动后挂载到根文件系统中的/usr 目录下。里面存放的是共享、只读的程序和数据,这表明/usr 目录下的内容可以在多个主机间共享。/usr 中的文件应该是只读的,关于其他主机的、可变的文件应该保存在其他目录下,比如/var。

/usr 目录通常包含如下内容,在嵌入式系统中,这些内容可以进一步精简。/usr 目录中的内容如表 7-9 所示。

表 7-9　/usr 目录中的内容

目　录	描　　述
bin	很多用户命令存放在这个目录下
include	C 程序的头文件,这在 PC 上进行开发时采用,在嵌入式系统中不需要
lib	库文件
local	本地目录
sbin	非必需的系统命令(必需的系统命令放在/sbin 目录下)
share	架构无关的数据
X11R6	X-Window 系统
games	游戏
src	源代码

9. /var 目录

与/usr 目录相反,/var 目录中存放的是系统运行时要改变的数据。如/var/log 子目录下存放了各种程序的 Log 文件。有些目录还可以与其他系统共享,如/var/mail、var/cache/man、/var/cache/fonts、/var/spool/news。var 目录存在的目的是把 usr 目录在运行过程中需要更改的文件或者临时生成的文件及目录提取出来,由此可以使 usr 目录挂载为只读的方式。

10. /proc 目录

这是一个空目录,常作为 proc 文件系统的挂接点。proc 文件系统是一个虚拟的文件系统,它没有实际的存储设备,里面的目录、文件都是由内核临时生成的,用来表示系统的运行状态,也可以操作其中的文件控制系统。

系统启动后,使用以下命令挂接 proc 文件系统(通常在/etc/fstab 下进行设置以自动挂接)。

11. /mnt 目录

用于临时挂接某个文件系统的挂接点,通常是空目录;也可以在里面创建一些空的子目录,比如/mnt/cdram、/mnt/hda1 等,用来临时挂接光盘和硬盘。

12. /tmp 目录

用于存放临时文件,通常是空目录。一些需要生成临时文件的程序要用到/tmp 目录,所以/tmp 目录必须存在并可以访问。

为了减少对 Flash 的操作,当在/tmp 目录上挂接内存文件系统时,可以使用如下命令:

```
mount -t tmpfs none /tmp
```

7.5.4 制作根文件系统

任务：掌握制作根文件系统的工具——Busybox 的使用方法。

所谓制作根文件系统，就是创建各种目录，并且在里面创建各种文件。比如在/bin、/sbin 目录下存放各种可执行程序，在/etc 目录下存放配置文件，在/lib 目录下存放库文件。本节将讲述如何使用 Busybox 来创建/bin、/sbin 等目录下的可执行文件。

1. Busybox 简介

Busybox 最初是由 Bruce Perens 在 1996 年为 Debian GNU/Linux 安装盘编写的。其目标是在一张软盘上创建一个可引导的 GNU/Linux 系统，可以用做安装盘和急救盘。

Busybox 包含了一些简单的工具，例如 cat 和 echo，还包含了一些较大、较复杂的工具，例如 grep、find、mount 以及 telnet(不过它的选项比传统的版本要少)；有些人将 Busybox 称为 Linux 工具里的瑞士军刀。

与一般的 GUN 工具集达到几 MB 相比，动态链接的 Busybox 只有几百 KB，即使静态链接也只有 1MB 左右。Busybox 按模块进行设计，可以很容易地加入、去除某些命令，或增减命令的某些选项。

在创建一个最小的根文件系统时，若使用 Busybox，只需要在/dev 目录下创建必要的设备节点，在/etc 目录下创建一些配置文件就可以了，如果 Busybox 使用动态链接，还要在/lib 目录下包含库文件。

Busybox 支持 uclibc 库和 glibc 库，对 Linux 2.2.x 之后的内核支持良好。

Busybox 的官方网站是 http://www.busybox.net，源码可以从 http://www.busybox.net/downloads 下载。

2. 编译/安装 Busybox

从 http://www.busybox.net/downloads/下载/busybox-1.7.0.tar.bz2，解压之后就可以看到源代码了。

Busybox 集合了几百个命令，在一般系统中并不需要全部使用。可以通过配置 Busybox 来选择这些命令、定制某些命令的选项、指定 Busybox 的链接方法(动态链接还是静态链接)、指定 Busybox 的安装路径。

(1) 配置 Busybox

在 Busybox-1.7.0 目录下执行 make menuconfig 命令即可进入配置界面。Busybox 将所有配置分类存放，表 7-10 中列出了这些类型，其中的“说明”是针对嵌入式系统而言的。

(2) 编译和安装 Busybox

编译和安装的步骤如下。

① 编译之前，先使用交叉编译器修改 Busybox 根目录的 Makefile。

修改前：

```
175 ARCH            ?=$ (SUBARCH)
176 CROSS_COMPILE     ?=
```

表 7-10　Busybox 配置选项分类

配置选项类型	说　明
Busybox Settings---> General Configuration	一些通用的设置，一般不需要设置
Busybox Settings---> Build Options	链接方式、编译选项等
Busybox Settings---> Debugging Options	调试选项，使用 Busybox 时将打印一些调试信息
Busybox Settings---> Installation Options	Busybox 的安装路径，不需要设置，可以在命令中指定
Busybox Settings---> Busybox Library Tuning	Busybox 的性能微调，比如设置在控制台上可以输入的最大字符个数，一般使用默认值即可
Archival Utilities	各种压缩、解压缩工具，根据需要选择相关命令
Coreutils	核心的命令，比如 ls、cp
Console Utilities	控制台相关命令，比如清屏命令 clear 等。可以提供一些方便，可以不使用
Debian Utilities	Debian 命令(Debian 是 Linux 的一种发行版本)，比如 which 命令可以用来显示一个命令的完整路径
Editors	编辑命令，一般都会选择 vi
Finding Utilities	查找命令，一般不用
Init Utilities	Init 程序的配置选项，比如是否读取 inittab 文件，使用默认配置即可
Login/Password Management Utilities	登录、用户账号、密码等方面的命令
Linux Ext2 FS Progs	ext2 文件系统的一些工具
Linux Module Utilities	加载、卸载模块的命令，一般都选中
Linux System Utilities	一些系统命令，比如显示内核打印信息的 dmesg 命令、分区命令 fdisk 等
Miscellaneous Utilities	一些不好分类的命令
Networking Utilities	网络方面的命令，可以选择一些方便调试的命令，比如 telnetd、ping、tftp 等
Process Utilities	进程相关命令，比如查看进程状态的命令 ps、查看内存使用情况的命令 free、发送信号的命令 kill、查看最消耗 CPU 资源的前几个进程的命令 top 等。为了方便调试，可以都选中
Shells	有多种 Shell，比如 msh、ash 等，一般选择 ash
System Logging Utilities	系统记录(log)方面的命令
Runit Utilities	实用程序，追踪最近和最常使用的程序
ipsvd Utilities	监听 TCP、DPB 端口，发现有新的链接时启动

修改后：

```
175 ARCH            ?=arm
176 CROSS_COMPILE      ?=arm-linux-
```

② 编译 Busybox，执行 make 命令。

③ 安装文件系统，执行 make CONFIG_PREFIX=install 命令。比如指定安装目录为/root/Myrootfs，则安装完成后，在/root/Myrootfs 目录下会看到生成的/bin、/sbin、/usr 目录和 linuxrc 文件。

④ 增加必要的文件。

到目前为止还没有得到一个完整可用的文件系统，必须要在这个基础上添加一些必要的文件，让它可以工作。

⑤ 制作文件系统镜像。

完成以上步骤后，可以将文件系统的镜像烧写到目标机上，内核运行时进行挂接，文件系统开始运行。

小结

本章讲述了构建 Linux 系统的全过程，包括嵌入式 Linux 的组成、开发主机和目标机之间的通信、Bootloader 的启动、Linux 内核的移植和配置、根文件系统的构建等内容。

第8章

基于Web的远程监控系统的设计实例

学习目标

通过本章的学习，应该掌握：

- 基于Web的远程监控系统的开发全过程
- 移植和配置嵌入式服务器的方法
- CGI程序设计方法
- 一个驱动程序的编写方法

基于 的远程监控系统简介

问题： 什么是嵌入式 Web 服务器？什么是基于 Web 的远程监控系统？建立一个基于 Web 的远程监控系统需要用到哪些技术？

重点： 嵌入式 Web 服务器的作用，使用嵌入式 Web 服务器构建的远程监控系统的架构，嵌入式 Web 服务器的典型应用。

内容： 嵌入式 Web 服务器和远程监控系统，基于 Web 的远程监控系统的典型应用。

本章将介绍一个简单的基于 Web 的远程监控系统的设计和实现的全过程，包括系统架构的设计、硬件设计和软件设计，重点介绍软件设计过程中使用的基础知识，并在此基础上逐步完成该系统的开发。

8.1.1 嵌入式 Web 服务器和远程监控系统

任务： 掌握嵌入式 Web 服务器的定义以及远程监控系统中嵌入式 Web 服务器所起的作用。

嵌入式 Web 服务器(Embedded Web Server，EWS)是指将 Web 服务器嵌入到现场测试和控制设备中，在相应的硬件平台和软件系统的支持下，使传统的测试和控制设备转变为具备了以 TCP/IP 为底层通信协议、Web 技术为核心的基于互联网的网络测试和控制设备。嵌入式 Web 服务器简化了传统服务器的系统结构，在嵌入式设备上同时实现信息传输和网络接口的功能。嵌入式 Web 服务器基于 HTTP 协议运作，有标准的接口形式和通信协议。它可以向任何接入它所在网络的合法用户提供统一的基于浏览器方式的操作和控制界面。Web 技术的开放性和平台独立特性能够降低开发难度，减少软件系统和通信系统的设计维护工作量，提高现场测试和控制设备的管理水平。

基于嵌入式 Web 服务器的远程控制系统可在嵌入式平台上完成对现场数据的实时采集，通过现场总线与数据网的互连，将服务器(Server)端的数据通过 TCP/IP 协议提供给远端的监控主机，即客户(Client)端。图 8-1 所示为一个典型的基于嵌入式 Web 服务器的远程监控系统。

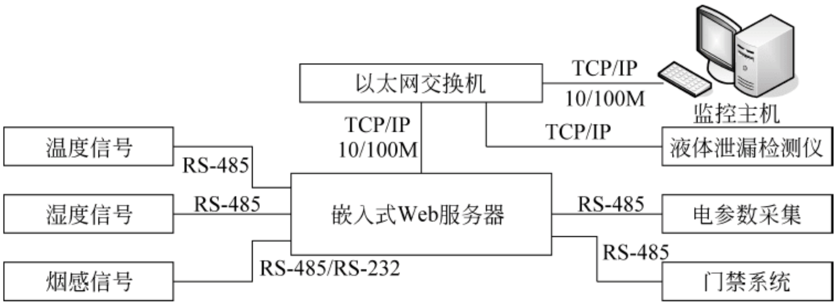


图 8-1 基于嵌入式 Web 服务器的远程监控系统结构图

其中,嵌入式 Web 服务器既是远程监控系统的中心节点,也是控制网络与数据网络进行互连的网关,通过 TCP/IP 协议将其连接到以太网上,监控主机则通过以太网(或 Internet)在远程实时地监视现场信号的动态变化,从而实现远程监控的目的。对于能够直接支持 TCP/IP 协议的现场检测设备,则可以将其直接连接到以太网上,如图 8-1 中的液体泄漏检测仪所示;对于采用其他总线标准的现场设备,则需要通过嵌入式 Web 服务器进行信号转换之后把现场总线连接到以太网上。

基于 Web 的远程监控系统的方案是:先搭建一个嵌入式 Web 服务器,服务器运行时,远程的客户可以直接登录,同时,在服务器的后台运行一个通信子程序,这个子程序通过 RS-232/RS-485 接口发送控制命令或接收运行数据,可见这种方案技术含量较高,但可以传输大量的数据。

在整个系统的实现过程中,嵌入式 Web 服务器起着十分重要的作用,使用嵌入式 Web 服务器的好处有以下几点。

(1) 远程监控终端仅需要安装浏览器即可,IE 或 Netscape 等软件大多由操作系统自带,无须开发专门的应用软件,可降低系统成本。

(2) 浏览器所在的监控终端平台与 Web 所在的服务器平台无关,监控终端可以采用多种操作系统,真正实现了跨平台。

(3) 操作界面简单统一,表达直观生动,用户无须经过专门培训。

(4) 易于扩展新的功能,系统升级仅需在 Web 服务器一端添加相应模块,与远程监控终端无关,可降低系统升级维护费用。

(5) 可提供分布式并行处理功能,基于 Web 的测控系统可构成一个多 CPU 协调工作的分布式测控系统,可并行处理多个测控指令。

8.1.2 基于嵌入式 Web 的远程监控系统应用

任务: 了解嵌入式 Web 远程监控系统的几种典型应用。

基于 Web 的远程监控系统对远程终端要求低(安装有浏览器的 PC 即可),再加上现有的 Internet 和宽带的普及,基于 Web 的远程控制方式将得到广泛的应用,可广泛地应用于工业设备远程监控、自动化农业、网络化信息家电、智能楼宇、远程安防监控系统等。下面列举一些嵌入式 Web 服务器的应用。

1. 在 UPS 监测系统中的应用

UPS(Uninterruptible Power Supply,不间断电源系统)作为网络系统中的保护设备,主要起到两个作用:其一是为计算机系统提供备用电源,目的是防止电网突然断电对重要文件数据造成损害;其二是消除电网供电上的“污染”(包括浪涌、波动、脉冲、噪声等),使计算机中的电子部件免受摧毁性损坏。而如何确保 UPS 系统的正常运作就成为一门新的课题。监测 UPS 中的电池状况、UPS 机房的环境温度、UPS 系统的负载情况将成为确保 UPS 系统正常工作不可或缺的一部分。

嵌入式 Internet 技术是以 Internet 技术和嵌入式技术的发展为基础的。该技术的出现使得各种家用设备接入网络成为可能。如果嵌入式设备提供 WWW 服务,则用户可以通过 Internet 远程监测各种设备。结合实际需要,利用 Internet 方便地实现对不同的 UPS 机房

进行统一监测和管理将成为大型 UPS 系统的基本要求。基于嵌入式 Web 服务器的 UPS 监测系统由两部分组成：测量网和嵌入式 Web 服务器。其中，测量网完成对 UPS 机房的监测并将测量结果传送到嵌入式 Web 服务器上；嵌入式 Web 服务器获取测量结果并完成 Web 服务器功能，使得用户可以远程访问测量结果。图 8-2 为其系统结构示意图。

图 8-2 UPS 监测系统结构示意图

2. 在网络视频监控系统中的应用

目前，以网络为基础的数字视频监控系统是视频监控系统发展的主流，而随着微处理器技术、计算机网络技术的进步，基于嵌入式 Web 的网络视频监控系统逐渐得到了人们的广泛关注，其主要原理是：嵌入式视频服务器采用嵌入式实时操作系统，内置嵌入式 Web 服务器，把摄像机传送过来的视频信号经高效压缩芯片压缩后，通过内部总线传送到内置的 Web 服务器中。用户在监控端可以直接通过浏览器观看 Web 服务器上的摄像机视频图像，授权用户还可以控制摄像机云台镜头的动作。图 8-3 为嵌入式视频监控系统的结构示意图。

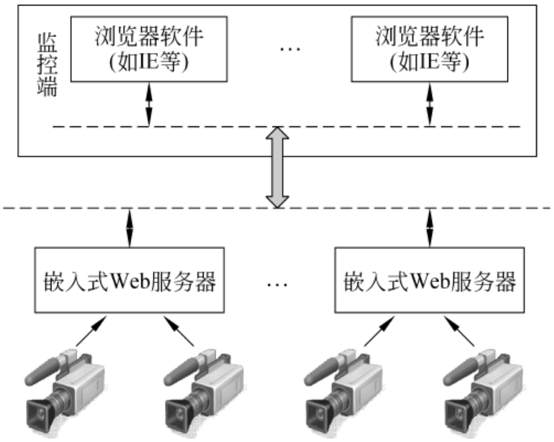


图 8-3 嵌入式视频监控系统结构示意图

嵌入式视频监控系统由摄像机、嵌入式 Web 服务器、传输网络和监控端组成。摄像机用来采集监控现场的视频；嵌入式 Web 服务器是整个监控系统的核心，包括硬件和软件两个部分，其主要功能有：为监控端提供 Web 访问页面，对监控端的访问进行有效性、安全性检查，响应监控端的请求，为监控端提供所需要的视频图像，接收监控端的控制信息，经过软、硬件转换后对摄像机进行控制，每个服务器有自己的 IP 地址，在监控端可以通过浏览器界面访问服务器；监控端的功能则是显示现场视频，并根据需要向服务器发送视频请求以及对摄像机的控制信号。

3. 在电力系统中的应用

电力系统是一个包含了电能生产、传输和使用的复杂系统，各种设备地域分布广泛、类型众多、数量巨大，对这些设备的监控、诊断和维护成为提高系统管理水平的重要内容。而

电力系统经过设备改造后，其硬件基础和信息网络的建设得到了极大的改善，这就为嵌入式 Web 服务器技术在电力系统中的应用打下了良好的基础。下面是嵌入式 Web 服务器技术在电力系统中的应用实例。

（1）基于嵌入式 Web 服务器技术的网络视频监控在电力系统中的应用

该系统主要应用于无人值守的变电站、电力局/电厂综合监控系统、现场生产调度指挥系统以及对灾害和突发事件的应急处理，以确保监控场所内设备的可靠运行及人员的安全。相对于传统的监控系统而言，它具有节省费用、即插即看、性能高、网络环境独立和设备可灵活接入等优点。

（2）基于嵌入式 Web 服务器技术的远程监测与故障诊断在电力系统中的应用

将 Web 技术与监测及故障诊断技术结合起来的策略有 3 种：设备端嵌入式 Web 服务器体系、基于服务的 Web 服务器体系和综合的多 Web 服务器的系统体系等。其中设备端嵌入式 Web 服务器体系把 Web 服务器放在设备处，与设备集成在一起，用户通过 Web 浏览器监测设备状态，同时可通过浏览器编程的方式来控制设备端的数据采集方式。通过合理分配计算任务，可以充分发挥客户机的强大计算能力，减轻设备端的计算负荷。恰当地把 Web 技术与设备的监测与故障诊断技术结合起来，将突破监测及故障诊断原来相对封闭的概念框架，是对设备监测与故障诊断技术的提升。

（3）基于嵌入式 Web 服务器技术的保护和控制设备在变电站自动化系统中的应用

将嵌入式 Web 服务器技术应用在变电站的保护和控制设备中将导致传统变电站自动化领域运行维护模式发生变革。通过 Web 浏览器获取嵌入式 Web 服务器中的系统实时信息，进而实现远程实时控制、调节和维护。基于嵌入式 Web 服务器技术的变电站自动化系统可以实现的功能包括：实时数据与历史数据动态发布功能、参数设置功能、远程实时控制功能、文件下载与上传功能、电子邮件告警功能、访问级别设置与权限认证功能、PPP 拨号上网等功能。

目前 ABB 公司开发的 REF542 plus 新一代综合保护和控制继电器中已经集成了嵌入式 Web 服务器技术，可以实现基于浏览器或当地控制装置进行实时控制，国内的东方电子信息产业股份有限公司开发的 DF 3600 系统也可实现类似的功能。由于电力系统安全运行的重要性，嵌入式 Web 服务器技术在现场监测和控制设备中的实际应用是逐步深入和不断扩展的。而随着 TCP/IP 技术在变电站通信系统中的应用，特别是将带有区分服务的宽带 TCP/IP 技术应用到设备层和间隔层间的通信后，可以满足实时性要求，能更充分地发挥嵌入式 Web 服务器技术的优势，提高远程监测和控制水平。

4. 网络机器人

在现代制造系统（柔性制造系统、计算机集成制造系统或现代集成制造系统）中，网络机器人已经成为一种像数控机床一样的必不可少的设备。因为人们能够把操作器搬离危险的作业现场，从而可以远程控制网络机器人在危险的环境中执行任务。

网络机器人控制系统包括以下 3 个部分。

（1）控制台：它是一个用于在远程对机器人进行控制的操作台，负责发送控制指令和接收从服务器端反馈来的信息，是网络机器人与控制人员进行交流的接口。

（2）控制服务器：它主要负责控制权的管理和相关信息的收发工作。可以实现用户跟踪及控制权的管理。

(3) 控制卡接口程序：用来向步进电机控制卡发送指令的部分。

5. 基于Web的机房环境设备的远程监控系统

随着计算机和网络技术的普及，计算机系统的数量与日俱增，计算机机房已成为各大型单位的信息枢纽。机房中的环境设备(如空调、UPS(不间断电源)、配电柜、消防设备等)为网络系统的安全运行提供了环境保障。同时，环境设备自身的安全运行也成为机房管理的重要内容之一。一旦机房的环境设备出现故障，就会直接影响计算机系统的正常运行，严重的还会造成机房内的相关设备损坏，甚至导致网络系统瘫痪。因此，对机房环境设备的运行状态进行实时监控，是保证机房设备安全运行的关键措施。图8-4为典型的基于Web的机房环境设备的远程监控系统的结构示意图。

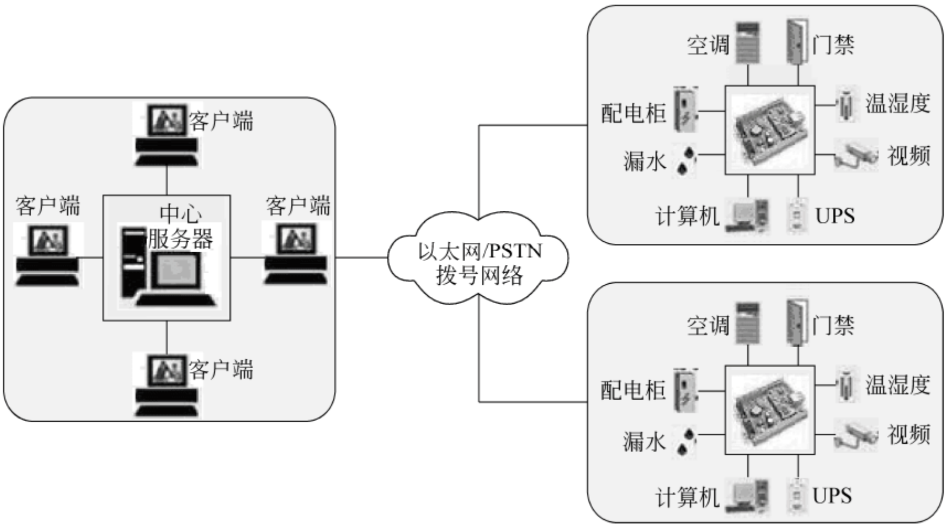


图8-4 基于Web的机房环境设备的远程监控系统的结构示意图

系统架构设计

问题：基于Web的远程监控系统的系统架构怎样构建？
重点：Web的远程监控系统的网络架构、硬件架构和软件架构。
内容：Web的远程监控系统的系统架构。

在嵌入式系统研发过程中，一般要先进行架构设计，综合考虑开发成本和生产成本，合理地选择嵌入式系统的结构形式，针对企业的技术积累情况，选择合适的技术解决方案。

8.2.1 网络架构

任务：掌握嵌入式Web服务器的定义以及远程监控系统中嵌入式Web服务器所起的作用。

基于Web的远程控制系统以嵌入式Web服务器为中心，通过Internet远程访问嵌入式Web服务器，嵌入式Web服务器通过现场总线控制各个节点，以达到远程监控的目的。

现场总线可以是 RS-485、CAN 或 ZigBee 等。任何一台装有浏览器的 PC 都可通过 Internet 访问嵌入式 Web 服务器。图 8-5 为基于 Web 的远程控制系统的网络示意图。

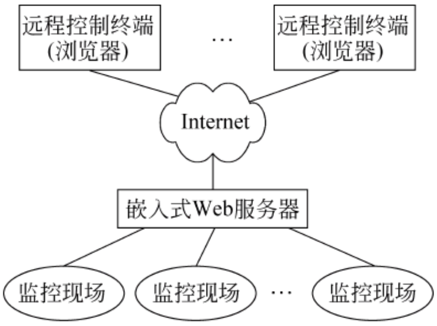


图 8-5　基于 Web 的远程控制系统的网络示意图

8.2.2　硬件架构设计

任务：了解基于 Web 的嵌入式远程监控系统的硬件设计方案。

1. 选择主控芯片

在硬件设计中，主控芯片的选择是设计的核心。主控芯片的选择可以从芯片的性能、供应情况、价格以及使用方的情况等方面来考虑。

三星 ARMS3C2440/2410 芯片在市面上比较流行，相关的资料及其软件方面的资料也很丰富，价格适中偏高，但考虑到基于这款芯片的软、硬件平台资源比较丰富，能够使用户很快学会使用，所以本章实例选择这款芯片作为主控芯片。

2. 基于 S3C2440A 的硬件设计方案

硬件系统选择上海双实科技有限公司的 SinoSys-M31 嵌入式教学实验系统作为 Web 服务器。该系统核心 CPU 采用三星公司主流 ARM9 处理器 S3C2440A，主频 400MHz，片上资源丰富，板上集成了与嵌入式系统相关的常用接口，能够满足 Web 服务器的功能需求。主要资源配置如表 8-1 所示。

表 8-1　基于 S3C2440A 的主要资源配置

序　号	名　称	描　　述
1	CPU	Samsung S3C2440A；400MHz 主频
2	ROM	2MB SST Nor Flash；64MB 三星 Nand Flash
3	RAM	64MB SDRAM，133MHz 刷新频率
4	LAN	1 个 10Mbps Ethernet，RJ-45 接口；1 个 10/100Mbps Ethernet，RJ-45 接口
5	SERIAL	1 个 DB9 标准从串口；1 个 DB9 标准主串口
6	USB	2 个 USB Host A 型接口(其中一个为主从复用，USB 1.1 协议)；一个 USB Slave B 接口(主从复用端口，USB 1.1 协议)
7	Audio	1 个音频接口，1 个音频输入口
8	RTC	外接 32.768kHz 的晶振
9	JTAG	1 个 20 针标准的 JTAG

续表

序　号	名　称	描　　述
10	IDE	标准 IDE 接口
11	LED	4 个可编程用户 LED
12	485	标准 485 两芯接口
13	Reset	1 个复位按键
14	Power	1 个开关电源(＋12V 供电)

8.2.3 软件架构设计

任务： 了解基于 Web 的嵌入式远程监控系统的软件设计方案，包括操作系统的选择和 Web 服务器方案的选择。

1. 操作系统的选择

嵌入式 Web 服务器应具有良好的性价比，服务器的操作系统可选用 Linux、Windows CE 等。

Windows CE 有良好的开发工具，但占用的系统资源比较多，提高了嵌入式 Web 服务器的成本。Linux 有很多的共享代码，可裁减性也比较好，占用的系统资源适中，但软件方面的工作量比较大。目前正在开发的嵌入式系统中，将近一半的项目都选用 Linux 作为嵌入式操作系统，因此，本章实例选择 Linux 操作系统。

2. Web 服务器方案

随着互联网应用的普及，越来越多的信息化产品需要接入互联网通过 Web 页面进行远程访问。嵌入式 Web 系统提供了一种经济、实用的互联网嵌入式接入方案。

在 Linux 操作系统中，常用的 Web 服务器有 3 个，分别是 httpd、thttpd 和 boa。httpd 是最简单的 Web 服务器，它的功能最弱，不支持认证，不支持 CGI。thttpd 和 boa 都支持认证、CGI 等，功能都比较全。

boa 是一个单任务的小型 HTTP 服务器，源代码开放，性能优秀，特别适合应用在嵌入式系统中。

CGI(Common Gateway Interface)是外部扩展应用程序与 Web 服务器交互的一个标准接口。按照 CGI 标准编写的外部扩展应用程序可以处理客户端浏览器输入的数据，从而完成客户端与服务器的交互操作。CGI 规范定义了 Web 服务器如何向扩展应用程序发送消息，在收到扩展应用程序的信息后如何进行处理等内容。通过 CGI 可以提供许多静态的 HTML 网页无法实现的功能，例如，搜索引擎、基于 Web 的数据库访问等。

进行综合考虑后，本章实例选择 boa＋CGI 的方案。

系统软件实现

问题： 移植嵌入式 Web 服务器包括哪些步骤？HTML 表单在基于 Web 的嵌入式系统中的作用是什么？CGI 是什么？CGI 程序和表单是什么关系？CGI 程序怎样处理表单提交的数据？

重点：嵌入式 Web 服务器的移植和配置，HTML 表单的设计，在表单中调用 CGI 程序的方法，表单向 CGI 程序提交数据的方法：get 和 post，CGI 程序设计。
内容：boa 服务器的移植和配置，HTML 表单的设计和使用，CGI 程序设计。

在嵌入式系统设计中，硬件设计占有的比重很大，由于选择了上海双实科技有限公司的 SinoSys-M31 嵌入式教学实验系统，其硬件配置在前面已有描述，该系统能够满足需求，在此不再重复。下面重点介绍系统软件的实现。

嵌入式 Web 服务器的软件开发主要包括以下 5 项内容。

(1) Bootloader 的移植。

(2) 嵌入式操作系统的移植和 TCP/IP 协议的剪裁。

(3) 嵌入式 Web 服务器的移植。

(4) 相关驱动程序的编写。

(5) CGI、数据采集、数据处理等相关应用程序的编写。

在第 7 章讲述了关于 Bootloader 的移植、嵌入式操作系统的移植和根文件系统的制作。本节主要讲述嵌入式 Web 服务器的移植、CGI、数据采集、数据处理等相关应用程序的编写以及相关驱动程序的编写。

8.3.1　嵌入式 Web 服务器的移植和配置

任务：掌握嵌入式 Web 服务器的移植和配置方法。

随着 Internet 技术的兴起，在嵌入式设备的管理与交互中，基于 Web 的应用成为目前的主流，这种程序结构也就是大家非常熟悉的 B/S 结构，即在嵌入式设备上运行一个支持脚本或 CGI 功能的 Web 服务器，能够生成动态页面，在客户端只需要通过 Web 浏览器就可以对嵌入式设备进行管理和监控，非常方便实用。本节主要介绍这种应用的开发和移植工作。

由于嵌入式设备资源一般都比较有限，并且也不需要能同时处理很多用户的请求，因此不使用 Linux 下最常用的服务器(如 Apache 等)，而需要使用一些专门为嵌入式设备设计的 Web 服务器，这些 Web 服务器在存储空间和运行时所占有的内存空间上都会非常适合于嵌入式应用场合。典型的嵌入式 Web 服务器有 boa(http://www.boa.org/)和 thttpd (http://www.acme.com /software/thttpd/)等，它们和 Apache 等高性能的 Web 服务器的主要区别在于，它们一般是单进程服务器，只有在完成一个用户请求后才能响应另一个用户的请求，而无法并发响应，但这在嵌入式设备的应用场合已经足够了。

本节主要讲述 boa 服务器的移植和配置。

要想移植 boa，首先要下载其源码，然后进行交叉编译、配置和测试，之后便可以移植了，具体步骤如下。

1. 下载源码

从 www.boa.org 下载 boa 源码，解压并进入源码目录的 src 子目录：

```
#tar  xzf  boa-0.94.13.tar.gz
#cd  boa-0.94.13/src
```

2. 交叉编译 boa

(1) 生成 Makefile 文件。运行 src 下的 configure 文件,可以生成 Makefile 文件:

```
# ./configure
```

(2) 修改 Makefile 文件。找到 CC=gcc 和 CPP=gcc -E,分别将其改为交叉编译器的安装路径:

```
CC=arm-linux-gcc
CPP=arm-linux-gcc -E
```

保存退出。

(3) 编译。得到可执行程序 boa:

```
#make
#arm-linux-strip boa                    //去掉文本信息,使 boa 变小
```

3. 配置 boa

为了能够在开发板上运行 boa,必须配置 boa 服务器的运行环境。首先在/etc 目录下建立一个/boa 目录,里面放入 boa 的主要配置文件 boa.conf。在 boa 源码目录下已有一个示例 boa.conf,可以在其上进行修改。

(1) Group 的修改

修改 Group nogroup 为 Group 0。

由于在/etc/group 文件中没有 nogroup 组,所以设成 0。另外在/etc/passwd 中有 nobody 用户,所以 User nobody 不用修改(注意,如果没有 nobody 用户,也需要将 User 设成 0)。

(2) ScriptAlias 的修改

指定 CGI 脚本的存放位置:修改 ScriptAlias/cgi-bin//usr/lib/cgi-bin/为 ScriptAlias/cgi-bin//var/www/cgi-bin/。

添加网页存放的位置:ScriptAlias /index.html /var/www/index.html。

(3) ServerName 的设置

修改#ServerName http://www.your.org.here/为 ServerName http://www.your.org.here/。

去掉注释,使 ServerName 行生效。若没有这一步的修改,boa 会出现异常而退出。提示 gethostbyname::No such file or directory,所以必须打开,其他保留默认设置即可。

4. 移植 boa

成功配置以后,还需要创建日志文件所在目录/var/log/boa,创建 HTML 文档的主目录/var/www,将静态网页存放在该目录下(可以将主机 /usr/share/doc/HTML/目录下的 index.html 文件和 img 目录复制到/var/www 目录下),创建 CGI 脚本所在目录/var/www/cgi-bin,将 CGI 的脚本存放在该目录下。另外还要将 mime.types 文件复制到/etc 目录下,通常可以从 Linux 主机的/etc 目录下直接复制。

假设 boa-0.94.13.tar.gz 的解压目录为/root/Myjob,则移植过程可归纳如下:

```
mkdir /etc/boa
```

```
cp /root/Myjob/boa/boa.conf /etc/boa/
mkdir /var/log
mkdir /var/log/boa
mkdir /var/www
cp /root/Myjob/boa/index.html /var/www/
cp -r /root/Myjob/boa/img /var/www/
mk dir /var/www/cgi-bin
cp /root/Myjob/boa/mime.types /etc/
cp /root/Myjob/boa /boa/
```

移植完成后，在后台运行 boa：

```
# ./boa&
[01/Feb/2030:15:58:00 +0000] boa: server version Boa/0.94.13
[01/Feb/2030:15:58:00 +0000] boa: server built Dec 21 2007 at 02:03:37
[01/Feb/2030:15:58:00 +0000] boa: starting server pid=37, port 80
```

出现上面的信息表示服务器运行正常。

5．测试 boa

（1）静态 HTML 网页测试

在目标平台上运行 boa，将主机与目标平台的 IP 设成同一网段，然后打开任意一个浏览器（Linux 或 Windows 下的都可以），输入目标平台的 IP 地址（http://192.168.2.23），出现图 8-6 所示的 BOA TEST 的欢迎网页，则静态 HTML 网页调试成功。

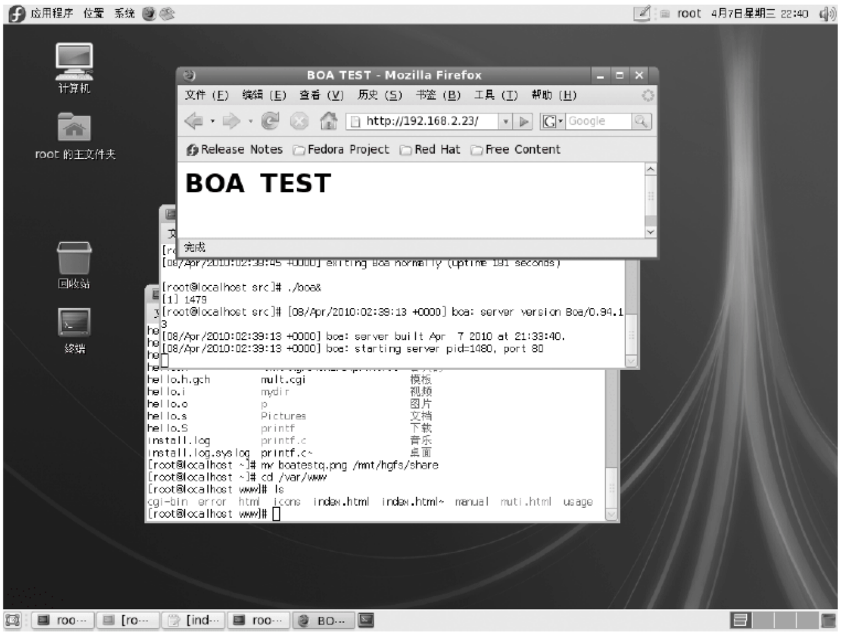


图 8-6　boa 静态网页测试

（2）CGI 脚本测试

编写一个简单的 CGI 程序 helloworld.c，代码如下：

```
#include< stdio.h>
int main()
```

```
{
  printf("Content-type: text/html\n\n");
  printf("<html>\n");
  printf("<head><title>Hello World!</title></head>\n");
  printf("<body>\n");
  printf("<h1>Hello World!</h1>\n");
  printf("</body>\n");
  printf("</html>\n");
  exit(0);
}
```

然后进行交叉编译，将得到的 helloworld.cgi 复制到 var/www/cgi-bin 目录下：

```
#arm-linux-gcc -o helloworld.cgi helloworld.c
```

在浏览器中输入 http://192.168.2.22/cgi-bin/helloworld.cgi，可以看到图 8-7 所示的页面，表示 CGI 测试通过。

图 8-7 CGI 脚本测试

6. boa 编译问题

boa 在编译过程中可能出现以下问题，下面给出解决方法。

问题：使用 toolchain 3.4.1 编译出现错误

```
arm-linux-gcc -g -O2 -pipe -Wall -I. -c -o util.o util.c
util.c:100:1: pasting "t" and "->" does not give a valid preprocessing token
make: *** [util.o] Error 1
```

解决：修改 src/compat.h

找到

```
#define TIMEZONE_OFFSET(foo) foo##->tm_gmtoff
```

修改成

```
#define TIMEZONE_OFFSET(foo) (foo)->tm_gmtoff
```

问题：执行 boa 出现 "gethostbyname::No such file or directory"

解决：需将 boa.conf 里的 ServerName 开头注释掉

问题：无法启动 boa，error log 显示 "boa.c:266.icky Linux kernel bug!:No such file"

解决：修改 src/boa.c，将底下的判断式注释掉：

```
/* if (setuid(0) != -1) {
```

```
DIE("icky Linux kernel bug!");
}*/
```

问题：无法启动 boa，error log 显示"boa.c:211 - getpwuid...略"

解决：修改 src/boa.c，将底下的两个判断式注释掉：

```
/* if (passwdbuf ==NULL) {
DIE("getpwuid");
}
if (initgroups(passwdbuf->pw_name, passwdbuf->pw_gid) ==-1) {
DIE("initgroups");
}*/
```

8.3.2 HTML 中表单的使用

任务：掌握 HTML 中表单的使用方法。

1. HTML 简介

HTML(HyperText Mark-up Language，超文本标记语言或超文本链接标记语言)是目前构成网页文档的主要语言。HTML 文件是包含一些标签的文本文件，这些标签指示 Web 浏览器如何显示页面。HTML 文件可以通过简单的文本编辑器来创建，文件名必须使用.htm 或者.html 作为文件扩展名。

例如，一个简单的页面 mypage.html 的内容如下：

```
<html>
<head>
<title>页面的标题</title>
</head>
<body>
<p>这是我的第一个页面。<b>这是粗体文本。</b></p>
</body>
</html>
```

HTML 文件中的第一个标签是<html>，通知浏览器这是 HTML 文件的开始点。文件中的最后一个标签是</html>，通知浏览器，这是 HTML 文件的结束点。位于<head>和</head>标签之间的文本是头信息。头信息不会显示在浏览器窗口中。<title>标签中的文本是文件的标题，标题会显示在浏览器的标题栏中。<body>标签中的文本是将被浏览器显示出来的文本。和标签中的文本将以粗体显示。用浏览器打开该网页，显示的内容如图 8-8 所示。

图 8-8 简单的 HTML 页面举例

2. HTML 标签

HTML 文档中的所有文本内容都是要显示在浏览器屏幕上的。HTML 标签包括影响文本表示和允许在文本中插入附加内容的信息，比如图形图像及导航链接。标签由左右两个尖括号

及其中的文本组成，文本通常不区分大小写。常用的 HTML 标签如表 8-2 所示。

表 8-2 HTML 标签

标　　签	属　　性
<!--This is a comment-->	注释，不显示
Test page	超链接
Help info	锚名称
Bold text	粗体
<body>Sample text</body>	文档主体
 Start of new line	另起一行
Emphasis text	强调文本
Smaller text	设置字体
<form action=resp. htm>…</form>	从……创建
Heading	标题
<head><title>My Doc</title></head>	文档头
<hr>New section	水平分隔线
<i>Italic</i>	斜体文本
<img src="graph. gif" alt="Test results">	图形图像
< input type="text" name="username">	表单的输入元素
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">	附加文档信息
New paragraph	换段标记
<pre>Program text</pre>	预格式化文本
<table><tr><td>Dat1</td><td>Dat2</td></tr></table>	表格数据
<title>Test document</title>	文档标题

3. HTML 中的表单

普通的 HTML 页面只能提供静态的信息给用户，如果要实现网页的交互就需要使用表单(form)了。

表单在 Web 网页中用于访问者输入信息，从而能采集客户端信息，使网页具有交互的功能。一般是将表单设计在一个 HTML 文档中，当用户输入完信息后进行提交(submit)，于是表单的内容就从客户端的浏览器传送到服务器上，经过服务器上的 ASP 或 CGI 等处理程序处理后，再将用户所需信息传送回客户端的浏览器上，这样网页就具有了交互性。

表单由一组相关联的标签组成，使用方法和 HTML 中的其他标签一样。在表单中提供了多种输入数据的工具，如文字输入区(Text)、下拉式菜单(Select)、复选框(CheckBox)、单选框(RadioButton)等。就目前所定义的标准中，form 的标签可以分为<input>、<select>以及<textarea>3 个大类。

(1) 表单的基本语法

```
<form name="form_name" method="method" action="url" enctype="value" target="target_win"
>
…
</form>
```

<form>标签定义了一个交互式表单，用户可以通过表单提交信息。表单的开始标记是<form>，结束标记是</form>。

(2) 表单的属性

<form>中具有下面几个属性。

① action="url"：指定一个方式来处理表单提交的数据。它可以是一个 URL 地址或一个电子邮件地址。

② method="method"：定义表单结果从浏览器传送到服务器的方法，一般有两种方法，即 get 和 post。

③ enctype = "value"：设置表单数据的编码方式，当其值为 ContentType 时，指明把表单提交给服务器时（当 method 属性值为 post 时）的互联网媒体类型（Internet Media Type），即 MIMETYPES。

④ onsubmit=script：指明当表单发送后执行的脚本程序。

⑤ onreset=script：指明表单被重置执行的脚本程序。

⑥ target="target_win"：表单提交时，指定信息的提交窗口。

⑦ accept-charset=charsets：指定一个字符编码列表（用户可以输入该列表），服务器可以根据该列表进行处理，该属性的值应该是以空格或者逗号隔开的字符集列表。如果表单没有 accept-charset 属性，默认值是 unknown，表示表单的字符集与包含表单的文档的字符集相同。

⑧ name=form_name：给表单指定一个名称。

总之，<form>标签指定了以下内容。

① 表单的界面（通过其他标签给出）。

② 用来处理表单提交的数据的程序（action 属性）。

③ 用户数据通过哪种方式传送到服务器（method 属性）。

④ 为了处理这个表单，字符编码方式必须能够被服务器接受（accept-charset 属性）。

4. input 标签

使用 input 标签可以通过表单提交用户信息。

(1) input 的基本用法

```
基本形式：<input type=输入类型>
```

type=[text password checkbox radio submit reset file hidden image button]：指定要建立的输入类型，默认值为 text。

根据不同的 type 属性值，输入字段拥有很多种形式。输入字段可以是文本字段、复选框、掩码后的文本控件、单选按钮、按钮等。input 是单标签，只有开始标签<input>，无结束标签。

(2) input 标签中 type 属性输入类型

type 属性中定义了以下 10 种输入类型。

① text：建立一个单行文本框。通过文本框提交的是输入的文字。

② password：类似于 text，但输入的文字通常用一种隐藏字符的方式显示。通过密码框提交的值是输入的文字（并非显示的字符）。

③ checkbox：建立一个复选框，该复选框有一个打开/关闭开关。当开关打开时，复选框的值是激活的；当开关关闭时，则这个值没有激活。复选框的值只有在复选框被选中时才提交。在同一个表单中的多个复选框可以使用同一个名称。

④ radio：建立一个单选框，该单选框有一个打开/关闭开关。当这个开关打开时，这个单选框的值是激活的；当开关关闭时，则这个值没有激活。单选框的值只有在开关打开的时候才提交。在同一个表单中的多个单选框可以有一个名称。然而，在同一个时刻只有一个单选框可以打开，在某个单选框设定为打开的时候，所有相关的按钮都是关闭的，因此，对于相关的单选框，只有一个值被提交。

⑤ submit：建立一个提交按钮。当这个按钮被用户激活时，表单中的所有数据将被提交到 form 标签的 action 属性指定的位置。

⑥ reset：建立一个复位按钮。当这个按钮被用户激活时，表单中所有用户输入的数据将被重设为它们的初始值。复位按钮的名称/值不与表单一起提交。

⑦ file：显示给用户一个文件名。当这个表单被激活时，此文件同其他用户输入的数据一样被提交到服务器。

⑧ hidden：建立一个不被浏览器显示的元素。但元素的名称和值与表单一起提交。这个类型通常被用来保存客户端/服务器之间可能被 HTTP 丢失的交换信息。

⑨ image：建立一个图像化的 submit 按钮。

⑩ button：建立一个按钮，没有默认动作。按钮动作通过联系客户端按钮事件来定义。value 特性的值被用做按钮的标签。

(3) input 中的属性

<input>中具有以下的属性。

① name＝cdata：为输入类型提供一个名称。如果提交表单，这个名称将与输入类型的当前值(value)成对提交。

② value＝cdata：指定输入类型的初始值。这个属性是可选项。

③ size＝cdata：指定这个 input 框能录入多少个字符。这个属性是可选项。

④ disabled：禁止此输入类型被用户输入。

⑤ readonly：表示只读(只能看到，不能修改)的输入域(框)。

⑥ maxlength＝integer：当输入类型为 text 或 password 时，这个属性指定了可以输入字符数的最大值。这个数值可以超过指定的 size，在这种情况下浏览器会提供一个滚动条。这个属性的默认值对用户输入的字符没有限制。

⑦ checked：当输入类型为 radio(单选框)时，这个属性用于指定单选框被选中。这个属性只适用于单选框。

⑧ src＝url：当输入类型为 image 时，这个属性用于指定显示在提交按钮上的图片的 URL 地址。

⑨ alt＝cdata：对于不能显示图像表单的浏览器，这个属性用于指定替代的文本。

⑩ usemap＝url：指出由<MAP>和<AREA>定义的映像的位置。

⑪ align＝[left center right justify]，定义元素及围绕的上下文字水平方向的对齐方式。可能的取值如下。

left：左对齐调整。

center：居中调整。

right：右对齐调整。

justify：两边填满调整。

⑫ tabindex＝integer：指定当前元素在当前文档制表键次序中的位置。这个值可以是正的，也可以是负的。

⑬ onfocus＝script：指定一个脚本程序，当此输入类型得到鼠标焦点时，这个脚本程序运行。这个属性可用在标签 label、input、select、textarea、button 中。

⑭ onblur＝script：指定一个脚本程序，当此输入类型失去鼠标焦点时，这个脚本程序运行。这个属性可用在标签 label、input、select、textarea、button 中。

⑮ onselect＝script：指定一个脚本程序，当用户在一个文字域中选择一些文本时，这个脚本程序运行。这个属性可用在标签 label、input、select、textarea、button 中。

⑯ onchange＝script：指定一个脚本程序，当此输入类型失去鼠标焦点或它的值在获得焦点时修改了，这个脚本程序运行。

⑰ accept＝cdata：这个属性只能与＜input type＝"file"＞配合使用，它规定能够通过文件上传进行提交的文件类型。当提交用户选择一个文件发送到服务器时，浏览器可以使用这个属性来过滤掉不符合要求的文件。

(4) 表单举例

编写 test_input.html 文件，内容如下：

```
<html>
<title>input 标签的例子</title>
<body>
<form action="http://www.knob.com/cgi-bin/hello.pl" method="post">
<p>
请输入用户名：<input type="text" name ="username">
<br>
你的密码：<input type="password" name="pwd">
<br>
你要发送的文件：
<input type="file" name="file">
<p>你爱吃的水果：</p>
<p>
<input type="checkbox" name="s1" value="s1" checked>
苹果
<input type ="checkbox" name="s2" value="s2" >
梨
<input type ="checkbox" name="s3" value="s3" >
香蕉
<input type ="checkbox" name="s4" value="s4" >
西瓜
</p>
<p>
<input type ="checkbox" name="s5" value="s5" >
哈密瓜
<input type ="checkbox" name="s6" value="s6" >
菠萝
```

```
<input type ="checkbox" name="s7" value="s7" >
荔枝
<input type ="checkbox" name="s8" value="s8" >
什么都不爱
</p>
<p align="center">
<input type="hidden" name="hide" value="someinfo">
<input type="submit" name="submit" value="提交数据">
<input type="reset" name="reset" value="重新填写">
</form>
</body>
</html>
```

该表单的运行结果如图 8-9 所示。

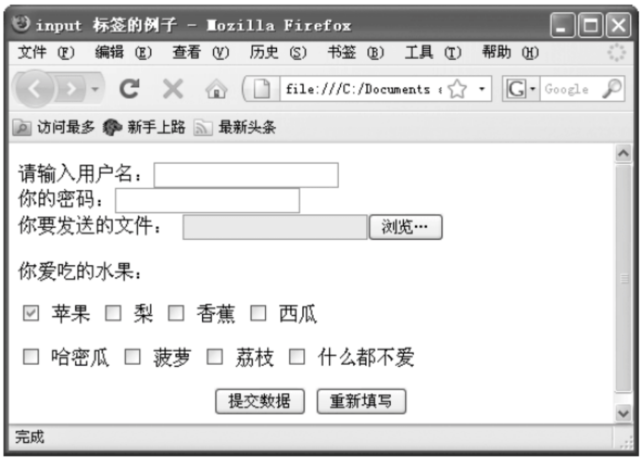


图 8-9 表单运行结果

注意 file 类型在显示时会多出一个“浏览”按钮，单击“浏览”按钮时，会打开一个选择文件的对话框，可以从中选择一个文件，当表单被提交时，这个文件会被一起提交。

5. select 标签

(1) select 标签的基本语法

<select>标签定义了一个下拉列表，用户可以在下拉列表中选择所需选项。其基本形式如下：

```
<select name=" ">
<option>
</select>
```

在 select 标签中还有一个内嵌的标签<option>，option 标签是单标签，只需开始标签，不需要结束标签。

(2) select 的属性

select 中具有以下几个属性。

① name＝cdata：为输入类型提供一个名称。如果表单被提交，这个名称将与输入类型的当前值（value）一起提交。

② size＝cdata：通知浏览器输入类型的初始值。这个属性是可选项。

③ multiple：用来设定列表中的项目允许多选。

④ disabled：禁止输入类型被用户输入。

⑤ tabindex＝integer：指定当前元素在当前文档制表键次序中的位置。这个值可以是正的，也可以是负的。

⑥ onfocus＝script：指定一个脚本程序，当此输入类型得到鼠标焦点时，这个脚本程序运行。这个属性可用在 label、input、select、textarea、button 标签中。

⑦ onblur＝script：指定一个脚本程序，当此输入类型失去鼠标焦点时，这个脚本程序运行。这个属性可用在 label、input、select、 textarea、button 标签中。

⑧ onselect＝script：指定一个脚本程序，当用户在一个文字域中选择了一些文本时，这个脚本程序运行。这个属性可用在 label、input、select、textarea、button 标签中。

⑨ onchange＝script：指定一个脚本程序，当此输入类型失去鼠标焦点或它的值在获得焦点时被修改了，这个脚本程序运行。

(3) option 的属性

option 中具有以下几个属性。

① selected：这个属性指定了此选项默认情况时是被选中的。

② disabled：禁止用户输入此输入类型。

③ value＝cdata：指定输入类型的初始值。

(4) select 标签使用举例

下面给出一个使用 select 标签的示例，其运行结果如图 8-10 所示。

图 8-10　select 标签使用举例

```
<html>
<title>select 标签的例子</title>
<body>
<form method="post" action="http://www.knob.com/cgi-bin/hello.pl">
<p>你的受教育程度：
<p><select name="edu">
```

```
<option value="edu1">没有上过学</option>
<option value="edu2">小学毕业</option>
<option value="edu3">初中毕业</option>
<option value="edu4">高中毕业</option>
<option value="edu5">大学毕业</option>
<option value="edu6">硕士毕业</option>
<option value="edu7">博士毕业</option>
</select>
<input type="submit" name="submit" value="提交">
</form>
</body>
</html>
```

6. textarea 标签

(1) textarea 的基本语法

textarea 标签定义了一个多行的文本输入域,使用户可以输入多行文本。textarea 的基本形式如下:

```
<textarea name=" " rows=" " cols=" " >
<textarea>
```

(2) textarea 的属性

textarea 中具有以下几个属性。

① name=cdata:为输入类型提供一个名称。如果表单被提交,这个名称将与输入类型的当前值(value)一起提交。

② rows=integer:指定可见的文字行的数目。当用户输入超过这个数量的行,内容超过可视区域时,浏览器会提供滚动条。

③ cols=integer:指定可视字符宽度。当用户输入的字符数超过这个值,内容超过可视区域时,浏览器会提供滚动条。

④ disabled:禁止此输入类型被用户输入。

⑤ readonly:禁止此输入类型被用户改变。

⑥ tabindex=integer:指定当前元素在当前文档制表键次序中的位置。这个值可以是正的,也可以是负的。

⑦ onfocus=script:指定一个脚本程序,当此输入类型得到鼠标焦点时,这个脚本程序运行。这个属性可用在 label、input、select、textarea、button 标签中。

⑧ onblur=script:指定一个脚本程序,当此输入类型失去鼠标焦点时,这个脚本程序运行。这个属性可用在 label、input、select、textarea、button 标签中。

⑨ onselect=script:指定一个脚本程序,当用户在一个文字域中选择了一些文本时,这个脚本程序运行。这个属性可用在 label、input、select、textarea、button 标签中。

⑩ onchange=script:指定一个脚本程序,当此输入类型失去鼠标焦点或它的值在获得焦点时被修改了,这个脚本程序运行。

(3) textarea 使用举例

下面通过一个例子说明 textarea 如何使用,其运行结果如图 8-11 所示。

```
<html>
<title>textarea 标签的例子</title>
<body>
<center>
<form action="http://www.baidu.com" method="post">
<p>你的留言:
<p>
<textarea rows="10" cols="30">
</textarea>
<p>
<input type="submit" name="submit" value="提交">
<form>
</center>
</body>
</html>
```

图 8-11　textarea 使用举例

7. 表单使用综合示例

现在综合前面所讲内容，编写 integration.html 文件，在这个表单中几乎用到了讲过的所有标签，内容如下：

```
<html>
<title>用户注册</title>
<body bgcolor="#ffffff">
<form method="post" action="http://www.baidu.com" enctype="multipart/form-data" name="
reg_user">
<h2 align=center">请填写你的详细资料</h2>
<p>你的姓名:
<input type="text" name="name">
你的 email:
<input type="text" name="email">
</p>
<p>你的密码:
```

```
<input type="password" name="password">
密码确认：
<input type="password" name="password2">
</p>
<p>你的性别：
<input type="radio" name="sex" value="male">
男
<input type="radio" name="sex" value="female">
女
<input type="radio" name="sex" value="no" checked>
不告诉你
</p>
<p>
你的生日：
<input type="text" name="year" maxlength="4" size="5" value="1975">
年
<input type="text" name="mon" size="3" maxlength="1" value="1">
月
<input type="text" name="day" size="3" maxlength="2" value="1">
日
</p>
<p>你的月收入：
<select name="money" size="1">
<option value="m1" selected>1000 以下</option>
<option value="m2">1000-3000</option>
<option value="m3">3000-5000</option>
<option value="m4">5000 以上</option>
</select>
</p>
<p>你的兴趣爱好：</p>
<p>
<input type="checkbox" name="music" value="music" checked>
音乐
<input type="checkbox" name="friend" value="friend">
交友
<input type="checkbox" name="travel" value="travel">
旅游
<input type="checkbox" name="book" value="book">
文学
</p>
<p>
<input type="checkbox" name="buy" value="buy">
购物
<input type="checkbox" name="sports" value="sports">
体育
<input type="checkbox" name="program" value="program">
编程
<input type="checkbox" name="game" value="game" checked>
电脑游戏
</p>
<p align="center">
```

```
<input type="submit" name="submit" value="提交资料">
<input type="reset" name="reset" value="重新填写">
</p>
</body>
</html>
```

运行 integration.html，结果如图 8-12 所示。

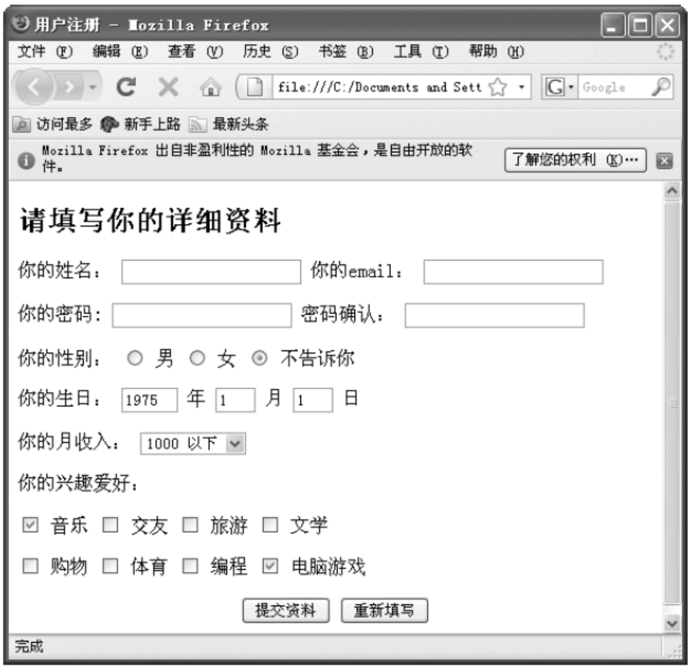


图 8-12　表单使用综合示例

8. HTML 超链接示例

下面是一个 HTML 超链接示例，主要展示如何通过多网页方式达到动态效果。在这里编写了 4 个 HTML 文件：digout00.html、digout01.html、digout10.html 和 digout11.html，这 4 个文件可以相互链接。

（1）digout00.html

```
<html><head></head>
<body bgcolor=#d0d0ff><font face=helvetica>
<h3><center>control</center></h3>
<center><table>
<tr valign=middle>
<td><a href="digout10.htm"><img src="sw0.gif"></a></td>
<td><a href="digout01.htm"><img src="sw0.gif"></a></td>
</tr><tr valign=middle>
<td align=center><img src="led0.gif"></td>
<td align=center><img src="led0.gif"></td>
</tr>
</table></center>
</body>
</html>
```

(2) digout01. html

```
<html><head></head>
<body bgcolor=#d0d0ff><font face=helvetica>
<h3><center>control</center></h3>
<center>
<table>
<tr valign=middle>
<td><a href="digout11.htm"><img src="sw0.gif"></a></td>
<td><a href="digout00.htm"><img src="sw1.gif"></a></td>
</tr><tr valign=middle>
<td align=center><img src="led0.gif"></td>
<td align=center><img src="led1.gif"></td>
</tr>
</table>
</center>
</body>
</html>
```

(3) digout10. html

```
<html><head></head>
<body bgcolor=#d0d0ff><font face=helvetica>
<h3><center>control</center></h3>
<center>
<table>
<tr valign=middle>
<td><a href="digout00.htm"><img src="sw1.gif"></a></td>
<td><a href="digout11.htm"><img src="sw0.gif"></a></td>
</tr><tr valign=middle>
<td align=center><img src="led1.gif"></td>
<td align=center><img src="led0.gif"></td>
</tr>
</table>
</center>
</body>
</html>
```

(4) digout11. html

```
<html><head></head>
<body bgcolor=#d0d0ff><font face=helvetica>
<h3><center>control</center></h3>
<center>
<table>
<tr valign=middle>
<td><a href="digout01.htm"><img src="sw1.gif"></a></td>
<td><a href="digout10.htm"><img src="sw1.gif"></a></td>
</tr><tr valign=middle>
<td align=center><img src="led1.gif"></td>
<td align=center><img src="led1.gif"></td>
</tr>
</table>
```

```
</center>
</body>
</html>
```

打开网页 digout00. htm，运行结果如图 8-13 所示。

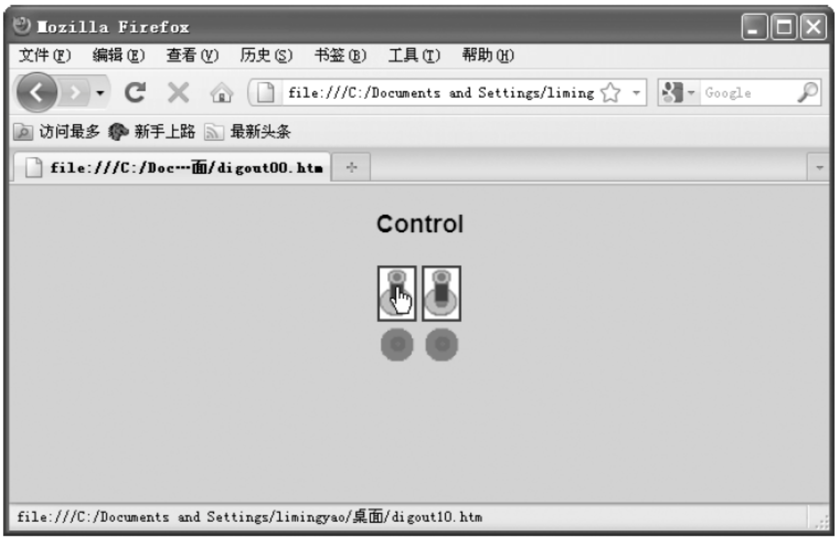


图 8-13　超链接示例(一)

单击超链接时，打开网页 digout10. htm，如图 8-14 所示。同理还可以单击其他的超链接。

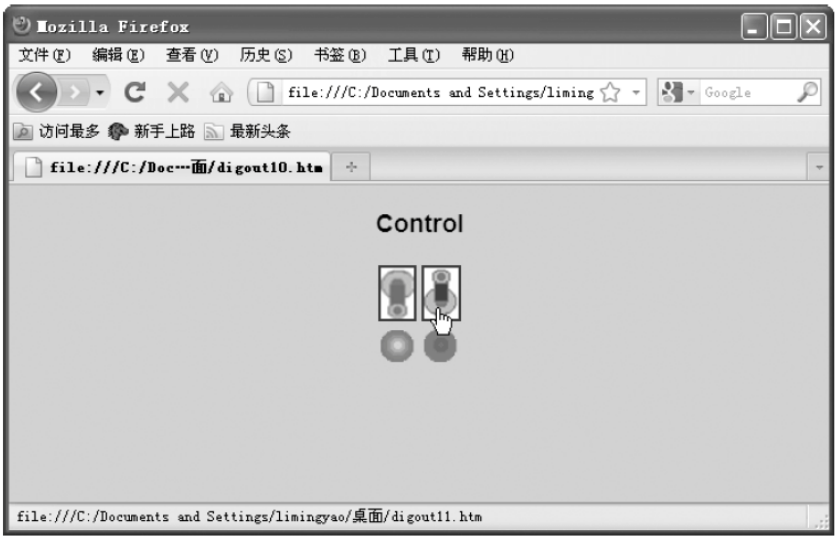


图 8-14　超链接示例(二)

8.3.3　CGI 程序设计

任务： 掌握 CGI 的工作原理和 CGI 程序设计方法。

CGI(Common Gateway Interface,外部扩展应用程序)可以称为CGI程序,是与WWW服务器交互的一个标准接口。按照CGI标准编写的外部扩展应用程序可以处理用户通过客户端浏览器输入的数据,从而完成客户端与服务器的交互操作。CGI规范定义了Web服务器应如何向扩展应用程序发送消息,在收到扩展应用程序的信息后又如何进行处理等内容。通过CGI可以提供许多静态的HTML网页无法实现的功能,比如搜索引擎、基于Web的数据库访问等。

1. 工作原理

(1) CGI的工作原理

CGI程序可以用来在Web网页内加入动态的内容。通过接口,浏览器能够发送一个可执行应用程序的HTTP请求,而不仅仅只是静态的HTML文件。服务器将运行指定的应用程序,读取与请求相关的信息,获得请求传过来的数值。例如用户填写并提交HTML表单数据后,浏览器将这些数据发送到Web服务器上。Web服务器接收这些数据并把这些数据传送给客户机指定的CGI程序进行处理,CGI程序运行结束后,生成HTML页面,Web服务器把CGI程序运行的结果再送回用户浏览器。HTML文件将会被用户的浏览器解释执行并将结果显示在浏览器上。图8-15为CGI程序工作的基本流程示意图。

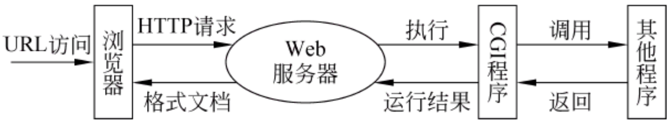


图8-15 CGI程序工作的基本流程示意图

CGI程序的处理步骤归纳如下。

第一步:通过Internet把用户请求传送到服务器。

第二步:服务器接收用户请求并交给CGI程序处理。

第三步:CGI程序将处理结果传送给服务器。

第四步:服务器将结果送回用户浏览器。

(2) URL编码

当用户提交一个HTML表单时,Web浏览器首先对表单中的数据以名字/值对的形式进行编码,并发送给Web服务器,然后由Web服务器传送给CGI程序。其格式如下:

```
name1=value1&name2=value2&name3=value3&name4=value4&...
```

其中,name是form表单中定义的input、select或textarea等标签,value是用户输入或选择的标签值,这种格式即为URL编码,程序需要对其进行分析和解码。要分析这种数据流,CGI程序必须首先将数据流分解成名字/值对,这可以通过在输入流中查找下面的两个字符来完成。

每当找到字符=时,标志着一个form变量名结束;每当找到字符&时,标志着一个form变量值结束。注意输入的最后一个变量的值不以&结束。

一旦名字/值对被分解后,还必须将输入的一些特殊字符转换成相应的ASCII字符,转换如下。

① 名字/值对之间用&分隔,空格用+代替。

② 名字与值对之间用＝分隔，如果参数未赋值，参数也同样出现在编码中，例如："姓名＝"。

一些特殊符号：&，%，＋，转化为带%的十六进制数：%NN。

(3) 环境变量

服务器与 CGI 程序交换信息的协作方式是通过环境变量实现的。例如，所有的机器都有一个 PATH 环境变量，当在当前目录找不到文件时就要查找 PATH 变量。环境变量是一个保存用户信息的内存区，当服务器收到一个请求后，它首先要收集它能得到的所有相关信息，并把它放入内存中。服务器不知道 CGI 程序到底需要哪些信息，所以它把这些信息都收集起来，以保证不遗漏重要信息。HTML 表单提交的数据传送到服务器后，CGI 程序将从环境变量中获取这些数据。常用的环境变量说明如表 8-3 所示。

表 8-3 常用的环境变量说明表

环境变量	说明
SERVER_PORT	服务器运行的 TCP 端口，通常 Web 服务器是 80
REQUEST_METHOD	post 或 get，取决于表单是怎样提交的
HTTP_ACCEPT	客户机支持的 MIME 类型清单
HTTP_USER_AGENT	提交表单的浏览器的名称、版本和其他平台性的附加信息
HTTP_REFERER	提交表单的文本的 URL
PATH_INFO	附加的路径信息，由浏览器通过 get 方法发出
PATH_TRANSLATED	在 PATH_INFO 中系统规定的路径信息
SCRIPT_NAME	指向这个 CGI 脚本的路径，是在 URL 中显示的(如/cgi-bin/thescript)
QUERY_STRING	脚本参数或者表单输入项(如果是用 get 方法提交的)，QUERY_STRING 包含 URL 中问号后面的参数
REMOTE_HOST	提交脚本的主机名，这个值不能被设置
REMOTE_ADDR	提交脚本的主机 IP 地址
REMOTE_USER	提交脚本的用户名。如果服务器的认证被激活，这个值可以设置
REMOTE_IDENT	如果 Web 服务器是在 ident (一种确认用户连接的协议)下运行的，提交表单的系统也在运行 ident，这个变量就含有 ident 返回值
CONTENT_TYPE	如果请求中包括数据，此变量指定数据类型的类别
CONTENT_LENGTH	对于用 post 方法提交的表单，标准输入口的字节数
SERVER_NAME CGI	脚本运行时的主机名和 IP 地址
SERVER_SOFTWARE	服务器的类型，如 CERN/3.0 或 NCSA/1.3
GATEWAY_INTERFACE	运行的 CGI 版本
SERVER_PROTOCOL	服务器运行的 HTTP 协议

其中，REQUEST_METHOD、QUERY_STRING、CONTENT_LENGTH 是 3 个非常重要的变量，它们用来表示数据是如何传送到 CGI 程序的。CGI 程序要做的事情就是从这 3 个变量中取出数据，进行下一步的处理。

(4) 提交 CGI 程序表单举例

下面通过一个简单的例子来说明利用表单提交数据和调用 CGI 程序的方法，以及形成的 URL 编码。编写表单 simple.html，内容如下：

```
<html>
<title>简单的表单</title>
<body>
<form action="/cgi-bin/cgi_get" method="get">
<p align="center">
参数 1:<input type="text" name="get_data1">
<br>
参数 2:<input type="text" name="get_data2">
<br>
<p align="center">
<input type="submit" name="submit" value="提交数据">
<input type="reset" name="reset" value="重新填写">
</form>
</body>
</html>
```

表单运行结果如图 8-16 所示。

图 8-16　提交 CGI 程序表单举例

表单中的 action="/cgi-bin/cgi_get"指明使用的 CGI 程序名为 cgi_get;method 属性指定提交数据的方法是 post 还是 get(后面将详细讲解),这里使用的是 get 方法。在这个示例中提交了两个数据,一个数据的名称是 get_data1;另一个数据的名称是 get_data2,它们的值由用户在网页中输入。

由 URL 的编码规则可知,最后形成的 URL 编码形式为:

```
name1=value1&name2=value2&name3=value3 ...
```

假如用户在 get_data1 域中输入的是 12,在 get_data2 域中输入的是 23,然后单击“提交”按钮,这个内容将被编码。最后形成的编码如下:

```
get_data1=12&get_data2=23&submit=%CC%E1%BD%CA%FD%BE%DD
```

2. 用 CGI 程序处理得到的数据

当 Web 浏览器将表单数据编码后,就传送给 Web 服务器,而 Web 服务器并非自己处理这些数据,而是先使用 CGI 程序来处理,然后再把 CGI 程序产生的处理结果返回给 Web 浏览器。CGI 程序通过标准输入(stdin)或环境变量来得到服务器的输入信息,并通过标准输出(stdout)向服务器输出信息。当 Web 服务器收到了由 Web 浏览器传来的数据时,就启

动<form>标签中 action 属性所指定的 CGI 程序。如果 method 属性值是 get,CGI 程序就从环境变量 QUERY_STRING 中获取数据;若 method 属性值是 post,CGI 程序就从标准输入(stdin)中获取表单数据。CGI 程序获取表单数据并进行处理后,还要向 Web 服务器返回指定的信息(如数据的处理结果等)。

(1) get 方法

get 方法是对数据的一个请求,被用于获得静态文档。当使用 get 方法时,CGI 程序将会从环境变量 QUERY_STRING 获取数据。为了处理客户端的请求,CGI 必须对 QUERY_STRING 中的字符串进行分析。当需要从服务器获取数据并且不改变服务器上的数据时,应该选用 get 方法;但是如果请求中包含的字符串超过了一定长度,一般是 1024B,那么就只能选用 post 方法。get 方法通过附加在 URL 后面的参数发送请求信息。这些参数将被放在环境变量 QUERY_STRING 中传给 CGI 程序。下面通过具体的示例来说明 get 方法的表单格式和 CGI 程序接收数据的过程。

下面是一段 HTML 代码,名称为 mult.html,内容如下:

```
<html>
<title>CGI GET test</title>
<body>
<h1>CGI GET test<h1><br>
<form method="get" action="cgi-bin/mult.cgi">
<input name="m" SIZE="5" >
<input name="n" SIZE="5" ><br>
<input type=submit value="确认">
</form>
</body>
</html>
```

运行结果如图 8-17 所示。

图 8-17　使用 get 方法发送数据

在图 8-17 所示的页面中输入数值,单击“确认”按钮,便可以将数据发送出去。为了对数据进行处理,比如将两数相乘,然后输出结果,还需要使用相应的 CGI 程序处理,在该例中使用的 CGI 程序为 mult.cgi,代码如下:

```
//mult.c
```

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  char * data;
  long m,n;
  /* 从 stdout 中输出,通知 Web 服务器返回的信息类型,一定要有两个空行 */
  printf("Content-type: text/html;charset=gb2312\n\n");
  printf("<html>\n");
  printf("<head><title>乘法结果</title></head>\n");
  printf("<body>\n");
  printf("<h1>乘法结果</h1>\n");
  printf("</body>\n");
  /* 从环境变量 QUERY_STRING 获取数据 */
  data=getenv("QUERY_STRING");
  if(data ==0)
  {
    printf("<p>错误!没有输入数据或者数据传输有问题");
    printf("</p>");
  }
  else if(sscanf(data,"m=%ld&n=%ld",&m,&n)!=2)/* 从 data 中提取数据给变量 m、n */
  {
    printf("<p>错误!输入数据非法。向表单中输入的必须是数字。");
    printf("</p>");
  }
  else
  {
    printf("<p>%ld 和%ld 的成绩是: %ld。",m,n,m*n);
    printf("</p>");
  }
  printf("</html>\n");
  exit(0);
}
```

前面已经提到,标准输出的内容就是要显示在浏览器中的内容。第一行的输出内容是必需的,也是一个 CGI 程序所特有的:printf("Content-type:text/html;charset=gb2312\n\n"),为了使 Web 服务器能正确理解所返回的是何种信息,CGI 规定在输出的信息体前加上一个头部信息,例如要返回 HTML 文档,则头部信息为 Content-type:text/html。这个输出用来作为 HTML 的文件头,另外,在 Content-type 的定义后面必须有两行空行。因为所有 CGI 程序的头部输出都是相近的,因而可以为其定义一个函数,以节省编程的时间,这是 CGI 编程常用的一个技巧。程序在后面调用了库函数 getevn 来得到 QUERY_STRING 的内容,然后使用 sscanf 函数把每个参数值读取出来,要注意 sscanf 函数的用法。

在程序被编译后,将其重命名为 mult. cgi 放在/cgi-bin/目录下,就可以被表单调用了。经过 CGI 程序处理后返回给浏览器的结果如图 8-18 所示。

(2) post 方法

当浏览器将数据从一个表单传送给服务器时一般采用 post 方法,而且在发送的数据超过 1024B 时也必须采用 post 方法。当使用 post 方法时,Web 服务器向 CGI 程序的标准输

图 8-18　CGI 程序处理结果

入 stdin 传送数据。发送的数据长度存放在环境变量 CONTENT_LENGTH 中。CGI 程序必须检查 REQUEST_METHOD 环境变量以确定是否采用了 post 方法，并决定是否要读取 stdin。下面通过一个例子介绍如何将数据通过表单提交给 CGI 程序，编写 post. html 表单，内容如下：

```
<html><head>
<meta http-equiv="Content-Language" content="zh-tw">
<title>CGI 示范网页</title></head>
<body>
<form method="post" action="cgi-bin/post.cgi">
<p>姓名：<input type="text" name="name" size="20"></P>
<p>密码：<input type="password" name="password" size="20"></P>
<p>性别：<input type="radio" value="1" checked name="sex">女 <input type="radio" name="
sex" value="2">男</P>
<p>兴趣：<input type="checkbox" name="interest" value="1">看书 <input type="checkbox"
name="interest" value="2">运动
<input type="checkbox" name="interest" value="3">逛街</P>
<p>学历：<select size="1" name="level">
<option selected value="1">高中</option>
<option selected value="2">大学</option>
<option selected value="3">研究生</option>
</select></P>
<p>地址：<textarea rows="5" name="address" cols="50"></textarea></p>
<p><input type="submit" value="Send" name="B1">
<input type="reset" value="Clear" name="B2"></p>
</form>
<body></html>
</form>
</body>
</html>
```

post. html 运行结果如图 8-19 所示。

表单中的 action="/cgi-bin/post. cgi"指明使用的 CGI 程序名为 post. cgi；method 属性指定提交数据的方法是 post。

当 CGI 程序接收到 WWW 服务器传来的数据时，必须要进行译码，才能得知用户当初所输入的数据。CGI 程序的译码就是窗体编码的反向操作，其原则如下。

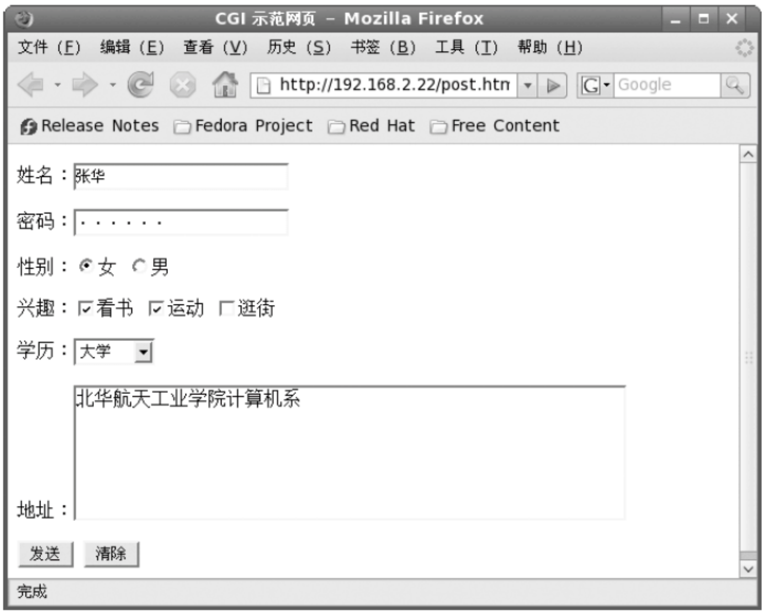


图 8-19 post.html 表单运行结果

① 先确定窗体使用 get 方法还是 post 方法来传送数据，可从 REQUEWT_METHOD 环境变量得知。

② 若窗体使用 get 方法传送，则读取 QUERY_STRING 环境变量来取得用户输入的数据。

③ 若窗体使用 post 方法传送，则读取 CONTENT_LENGTH 环境变量取得用户输入数据的长度后，再由标准输入设备取得这个长度的数据。

④ 将每个配好对的字段名称用＝输入值分离出来，取出配对数据之间的 & 符号。

⑤ 将＋号置换成空格，将%xx 十六进制码置换成原来的字符。

下面就是处理这个表单数据的 CGI 程序 post.c，该程序完成了上面的译码工作，内容如下：

```
//post.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct name_value_set
{
  char name[250];                //窗体字段名称最长为 250 个字符
  char value[250];               //窗体字段值最长为 250 个字符
}nv_set;
nv_set nv[200];                  //最多可以处理 200 个窗体字段

int get_input();
void makespace(char * s);
char * split(char * s,char stop);
char * convert(char * s);
```

```
int hexa(char c);

int main()
{
  int i,count;

/* 取得用户输入的数据,数据会存放在 nv 数组中 */
/* count 为窗体移码后实际的字段数 */
  count=get_input();

/* 输出 HTTP 标头 */
  printf("Content-type: text/html;charset=gb2312\n\n");

  printf("<html><head></head><body>\n");
  printf("<h3>窗体移码结果如下:  </h3>\n");

  /* 输出用户输入的字段名称及字段值 */
  for(i=0;i<count;i++)
  printf("<p>name=%s,value=%s\n</p>",nv[i].name,nv[i].value);
  printf("</body><html>\n");
  return 0;
}

/将窗体数据移码,并将译码结果存入 nv 数组中 */
int get_input()
{
  char * method;
  char * my_data=0;
  char * tmp_ptr, * tmp;
  int data_len;
  int i;
  method=getenv("REQUEST_METHOD");
  /* 窗体以 get 方法传送数据 */
  if(strcmp(method,"get")==0)
  {
    tmp_ptr=getenv("QUERY_STRING");
    data_len=strlen(tmp_ptr);
    /* 窗体编码数据存放在 my_data 中 */
    my_data=(char *)malloc(sizeof(char) * (data_len+1));
    strcpy(my_data,getenv("    QUERY_STRING"));
    my_data[data_len]='\0';
}
/* 窗体以 post 方法传送数据 */
if(strcmp(method,"post")==0)
{
  data_len=atoi(getenv("CONTENT_LENGTH"));
  /* 窗体编码数据存放在 my_data 中 */
  my_data=(char * )malloc(sizeof(char) * (data_len+1));
  fread(my_data,1,data_len,stdin);
}
i=0;
```

```
while(my_data[0]!='\0')
{
  tmp=split(my_data,'=');          //分离数据对,取得字段名
  makespace(tmp);                  //还原空格符
  tmp=convert(tmp);                //还原十六进制码
  strcpy(nv[i].name,tmp);          //结果存入 nv 结构的 name 成员中
  tmp=split(my_data,'&');          //还原十六进制码
  makespace(tmp);                  //还原空格符
  tmp=convert(tmp);                //还原十六进制码
  strcpy(nv[i].value,tmp);         //结果存入 nv 结构的 value 成员中
  i++;
}
return i--;                        //返回实际解码的字段数
}

/*还原空格符*/
void makespace(char *s)
{
  int i,len;
  len=strlen(s);
  for(i=0;i<len;i++)
  if(s[i]=='+') s[i]=' ';
}

/*分离数据对*/
char *split(char *s,char stop)
{
  char *data;
  char *tmp;
  int i,len,j;
  len=strlen(s);
  tmp=s;
  data=(char *)malloc(sizeof(char)*(len+1));
  for(i=0;i<len;i++)
  {
    if(s[i]!=stop) data[i]=s[i];
    else {i+=1;break;}
  }
  data[i]='\0';
  for(j=i;j<len;j++) s[j-i]=tmp[j];
  s[len-i]='\0';
  return data;
}

/*还原十六进制码*/
char *convert(char *s)
{
  int x,y,len;
  char *data;
  len=strlen(s);
  data=(char*)malloc(sizeof(char)*(len+1));
```

```
    y=0;
    for(x=0;x<len;x++)
    {
      if(s[x]!='%')
    {
      data[y]=s[x];
      y++;
    }
    else
    {
      data[y]=(char)(16*hexa(s[x+1])+hexa(s[x+2]));
      y++;
      x=x+2;
    }
    }
    data[y]='\0';
    return data;
  }

  /*将字符转换为十六进制码*/
  int hexa(char c)
  {
    switch (c)
    {
      case '0':return 0;
      case '1':return 1;
      case '2':return 2;
      case '3':return 3;
      case '4':return 4;
      case '5':return 5;
      case '6':return 6;
      case '7':return 7;
      case '8':return 8;
      case '9':return 9;
      case 'A':return 10;
      case 'B':return 11;
      case 'C':return 12;
      case 'D':return 13;
      case 'E':return 14;
      case 'F':return 15;
    }
  return 0;
  }
```

编译 post.c，将目标可执行程序命名为 post.cgi，将其存入/var/www/cgi-bin 目录下，在图 8-19 所示的表单中，填写表单内容，单击“发送”按钮，数据将会发送给 post.cgi，数据经 CGI 程序处理后传送到浏览器，运行结果如图 8-20 所示。

3. 用 C 语言进行 CGI 程序设计

CGI 程序是一种在 WWW 服务器上运行的程序，主要用于处理用户通过表单输入的信

图 8-20 post 提交数据示例

息，在服务器上产生相应的作用，或把处理结果返回给浏览器。CGI 程序可以用任何程序设计语言编写，如 Shell 脚本语言、Perl、Fortran、Pascal、C 语言等。

下面是一个用 C 语言编写的简单的 CGI 程序，它将 HTML 中的表单信息直接输出到 Web 浏览器上。

```
#include< stdio.h>;
#include< stdib.h>;
main()
{
  int i,n;
  printf("Content-type:text/plain\n\n");
  n=0;
  if(getenv("CONTENT_LENGTH"))
  n=atoi(getenv("CONTENT_LENGTH"));
  for(i=0;i<n;i++)
  putchar(getchar());
  putchar('\n');
  fflush(stdout);
}
```

下面对此程序进行简要的分析。

```
printf("Content-type:text/plain\n\n");
```

此行通过标准输出将字符串"Content-type:text/plain\n\n"传送给 Web 服务器。它是一个 MIME 头信息，通知 Web 服务器随后的输出是纯 ASCII 文本的形式。注意在这个头信息中有两个新行符，这是因为 Web 服务器需要在实际的文本信息开始之前先看见一个空行。

```
if(getenv("CONTENT_LENGTH"))
    n= atoi(getenv("CONTENT_LENGTH"));
```

此行首先检查环境变量 CONTENT_LENGTH 是否存在。Web 服务器在调用使用 post 方法的 CGI 程序时会设置此环境变量,它的文本值表示 Web 服务器传送给 CGI 程序的输入字符数目,因此使用函数 atoi()将此环境变量的值转换成整数,并赋给变量 n。注意 Web 服务器并不以文件结束符来终止它的输出,所以如果不检查环境变量 CONTENT_LENGTH,CGI 程序就无法知道什么时候输入结束。

```
for(i=0;i<n;i++)
putchar(getchar());
```

此行表示从 0 开始一直循环到(CONTENT_LENGTH-1)次,将标准输入中读到的每一个字符直接复制到标准输出,也就是将所有的输入以 ASCII 的形式回送给 Web 服务器。

通过此例可将 CGI 程序的一般工作过程总结如下。

(1) 通过检查环境变量 CONTENT_LENGTH 确定有多少输入。

(2) 循环使用 getchar()或者其他文件读函数得到所有的输入。

(3) 以相应的方法处理输入。

(4) 通过"Content-type:"头信息将输出信息的格式通知给 Web 服务器。

(5) 通过使用 printf()或者 putchar()或者其他的文件写函数,将输出传送给 Web 服务器。

总之,CGI 程序的主要任务就是从 Web 服务器得到输入信息,进行处理,然后将输出结果再返回给 Web 服务器。

在上面的例子中是通过使用 MIME 头信息"Content-type:text/plain\n\n"和 printf()、putchar()等函数调用来输出纯 ASCII 文本给 Web 服务器的。实际上,也可以使用 MIME 头信息"Content-type:text/html\n\n"来输出 HTML 源代码给 Web 服务器。发送这个 MIME 头信息给 Web 服务器后,Web 浏览器就会将随后的文本输出当成 HTML 源代码,在 HTML 源代码中可以使用任何 HTML 结构,如超链接、图像、form,及对其他 CGI 程序的调用。也就是说,可以在 CGI 程序中动态产生 HTML 源代码输出,下面给出一个简单的例子,代码如下:

```
#include<stdio.h>
#include<string.h>
main()
{
  printf("Content-type:text/html\n\n");
  printf("<html>\n");
  printf("<head><title>An HTML Page From a CGI</title></h ead>\n");
  printf("<body><br>\n");
  printf("<h2>This is an HTML age generated from with in a CGI program.</h2>\n");
  printf("<hr><p>\n");
  printf("<ahref=../output.html#two"><b>Go back to out put.html page </b></a>\n");
  printf("</body>\n");
  printf("</html>\n");
  fflush(stdout);
}
```

上面的 CGI 程序简单地用 printf()函数来产生 HTML 源代码。注意,在输出的字符串

中如果有双引号，在其前面必须有字符\，这是因为整个 HTML 代码串已经在双引号内，所以 HTML 代码串中的双引号必须用\来转译。

设备驱动程序设计

问题：设备驱动程序的功能是什么？Linux 系统中的驱动程序包括哪些部分？
重点：Linux 驱动程序组成，LED 驱动程序的设计和加载。
内容：Linux 驱动程序设计，字符驱动程序设计，LED 驱动程序设计。

设备驱动程序是操作系统内核和机器硬件之间的接口，它为应用程序屏蔽了硬件的细节，主要完成以下功能。

（1）探测设备和初始化设备。

（2）从设备接收数据并提交给内核。

（3）从内核接收数据传送到设备。

（4）检测和处理设备错误。

8.4.1 Linux 下的驱动程序设计基础

任务：了解 Linux 驱动程序的组成及各部分的作用。

Linux 设备驱动程序是内核与硬件之间的一个软件接口，可以通过两种方式将其集成到内核中：一是将其直接编译和静态链接到内核；二是通过 Linux 可加载模块（LKM）机制，将其编写成一种目标格式，实现为可动态加载和卸载的驱动模块。前者用户可随时调用，而无须安装，但增加了内核占用空间，并且更新需重编内核、重启系统；后者使用前必须先加载，但是更节省资源且灵活，也是通常采用的设备驱动程序设计方式。

1. Linux 设备类型

Linux 设备驱动程序有 3 种类型：字符设备、块设备和网络设备。

字符设备一次 I/O 操作存取数据量不固定，只能顺序存取，如鼠标、磁盘驱动器等设备。

块设备一次 I/O 操作以固定大小的数据块为单位，如硬盘、软驱等。其中，字符设备不经过系统的快速缓冲，而块设备经过系统的快速缓冲。

网络设备是经过特殊处理的，它没有对应的设备文件，Linux 使用套接口（socket），以文件 I/O 方式提供对网络数据的访问。其中与文件系统相关的两种类型是字符设备和块设备。

2. Linux 设备驱动程序的组成

不管是何种类型，从结构上看，整个驱动程序可分为驱动程序初始化、独立于设备的接口和硬件 I/O 共 3 个部分。如图 8-21 所示，驱动程序初始化部分负责将设备驱动程序装载到内核或从内核中卸载等；独立于设备的接口是设备驱动程序和文件系统连接的桥梁；而硬件 I/O 用于具体实现各种 I/O 操作。

从程序实现角度看，设备驱动程序也可分为以下 3 个部分。

(1) 自动配置和初始化子程序

负责检测所要驱动的硬件设备是否存在并能正常工作，如果该设备正常，则对这个设备及其相关的设备驱动程序需要的软件状态进行初始化。

(2) 服务于 I/O 请求的子程序

其主要是 file operations 结构的各个入口点的实现。这部分的实现支持文件系统调用(如 open、close、read、write 等)。

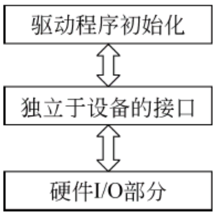


图 8-21　驱动程序组成

(3) 中断服务子程序

在 Linux 系统中并不是直接从中断向量表中调用设备驱动程序的中断服务子程序的，而是由 Linux 系统接收硬件中断，再由系统调用中断服务子程序。

3. 设备文件

Linux 把设备均作为文件来对待。这些文件一般称为设备文件，它使用户或应用程序可按操作普通文件的方式进行硬件设备访问控制。在 Linux 中，设备驱动程序是作为文件系统的一个模块存在的。它向下负责和硬件设备的交互，向上通过一个通用的接口挂接到文件系统上，从而和系统的内核等联系起来，管理和控制各种设备，是软件和硬件设备的一个抽象层。

设备文件的属性包括文件名、设备类型、主设备号、次设备号。主设备号是与驱动程序一一对应的。次设备号用来区分使用同一个驱动程序的个体设备。可以使用 major()函数获得主设备号，minor()函数获得次设备号。与普通的目录和文件一样，对设备的操作也是通过对文件操作的 file_operations 结构体来调用驱动程序的设备服务子程序。作为实现驱动程序的最重要的数据结构，它为 Linux 提供的服务于 I/O 请求的子程序的代码实现提供了一系列入口点，它们在设备驱动程序初始化的时候向系统进行登记，以便系统在适当的时候调用。file_operations 结构如下：

```
struct file_operations {
struct module * owner;
loff_t (* llseek) (struct file * , loff_t, int);
ssize_t (* read) (struct file * , char * , size_t, loff_t * );
ssize_t (* write) (struct file * , const char * , size_t, loff_t * );
int (* readdir) (struct file * , void * , filldir_t);
unsigned int (* poll) (struct file * , struct poll_table_struct * );
int (* ioctl) (struct inode * , struct file * , unsigned int, unsigned long);
int (* mmap) (struct file * , struct vm_area_struct * );
int (* open) (struct inode * , struct file * );
int (* flush) (struct file * );
int (* release) (struct inode * , struct file * );
int (* fsync) (struct file * , struct dentry * , int datasync);
int (* fasync) (int, struct file * , int);
int (* lock) (struct file * , int, struct file_lock * );
ssize_t (* readv) (struct file * , const struct iovec * , unsigned long, loff_t * );
ssize_t (* writev) (struct file * , const struct iovec * , unsigned long, loff_t * );
ssize_t (* sendpage) (struct file * , struct page * , int, size_t, loff_t * , int);
unsigned long (* get_unmapped_area) (struct file * , unsigned long, unsigned long, unsigned
long, unsigned long);
```

```
};
```

在 Linux 中，一个设备在使用之前必须向系统进行注册，设备注册是在设备初始化时完成的。对于可加载的内核驱动程序模块，有两个主要接口函数：一个用于在模块加载时注册服务和申请资源，如 init module()；另一个用于在模块卸载时清除由前者所做的工作，从而使内核模块可以安全地卸载，如 cleanup module()。编译后，root 用户执行 insmod 命令加载模块时调用前一个函数，执行 rmmod 命令卸载模块时调用后一个函数。

8.4.2 基于 Linux 2.6 内核的设备驱动程序举例

任务：了解一个简单的字符设备驱动程序的设计内容，及模块的加载和卸载方法。

Linux 2.6 内核设备驱动程序的统一框架定义了各种即插即用的硬件接口，子系统可通过这些接口与各个驱动程序进行通信。若想对已有模块进行编译，并将其加载到 2.6 内核，必须先完成一些基本的结构调整：用户需要为 MODULE_LICENSE()宏增加一个示例，例如 MODULE_LICENSE("GPL")。否则，当用户利用此类结构加载模块时，在标准输出设备和系统日志上会显示一个坏模块的出错信息。这种 2.4 内核以后的版本才引入的宏，可以将模块定义为获得 GPL Version 2 或更新版本许可的模块。其他有效值还有"Proprietary"、"GPL v2"、"GPL and additional rights"、"Dual BSD/GPL"(选择 BSD 或 GPL 许可)以及"Dual MPL/GPL"(选择 Mozilla 或 GPL 许可)。

为了便于参考，下面给出一个具体设备的驱动程序编写示例。编写一个名为 mydriver 的简单字符设备驱动程序，该驱动程序以可加载的模块方式进行编译，这样可以免去重新编译内核的工作。具体步骤如下。

(1) 编写简单的字符设备驱动程序 mydriver.c：

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/moduleparam.h>

static int __init mydriver_init(void);                        //设备初始化函数
static void __exit mydriver_exit(void);                       //设备注销函数
static int mydriver_open(struct inode *, struct file *);      //打开设备函数
static int mydriver_release(struct inode *, struct file *);   //释放设备函数
static ssize_t mydriver_read(struct file* ,char* ,size_t ,loff_t* );
                                                              //读设备函数
static ssize_t mydriver_write(struct file *, const char *, size_t, loff_t *);
                                                              //写设备函数

#define DEVICE_NAME "mydriver"                                //定义设备名称
static int major;                                             //定义设备主设备号

MODULE_LICENSE("GPL");

/* 定义对设备所进行的操作 */
```

```
static struct file_operations mydriver_fops={
        .read=mydriver_read,                             //读操作
        .write=mydriver_write,                           //写操作
        .open=mydriver_open,                             //打开操作
        .release=mydriver_release,                       //释放操作
};

/* 设备初始化函数的实现/
static int __init mydriver_init(void)
{
        major=register_chrdev(250, DEVICE_NAME, &mydriver_fops);    //注册设备
     if (Major < 0) {//注册失败
     printk ("Registering the character device failed with %d\n", Major);
     return Major;}
     return 0;//成功
}

/* 设备注销函数的实现 * /
static void __exit mydriver_exit(void)
{
unregister_chrdev(Major, DEVICE_NAME);                   //注销设备
}

/* 打开设备函数的实现 * /
static int mydriver_open(struct inode * inode , struct file * file)
{
printk("device open sucess!\n");
return 0;
}

/* 释放设备函数的实现/
static int mydriver_release(struct inode* inode, struct file* filp)
{
printk("device release\n");
return 0;
}

/* 读设备函数的实现/
static ssize_t mydriver_read(struct file* filp,char* buf,size_t count,loff_t* fpos)
{
int i;
/* 验证用户内存空间地址是否合法 * /
if(access_ok(VERIFY_WRITE,buf,count)==-EFAULT)
  return -EFAULT;
for(i=count; i>0; i--)
{
__put_user(1, buf);/* 从内核空间向用户空间分配 ASCII 码值为 1 的字符 * /
buf++;
}
return count;
}
```

```
/* 写设备函数的实现,空操作/
static ssize_t mydriver_write(struct file * filp,const char * buf,size_t count,loff_t *
fpos)
{
return count;
}

module_init(mydriver_init);                    //使用 mydriver_init 函数初始化设备
module_exit(mydriver_exit);                    //使用 mydriver_exit 函数初始化设备
```

由于该字符设备初始化时没有事先为字符设备申请内存空间,因此定义 mydriver_read()函数时,无法设计从内核空间向用户空间复制数据的操作。这里仅通过_put_user()函数从内核空间向用户空间分配 ASCII 码值为 1 的字符。

(2) 编写 Makefile 文件,目的是将驱动程序编译为内核的模块,内容如下:

```
PWD=$(shell pwd)
KERNELDIR?=/usr/src/kernels/2.6.23.1-42.fc8-i686/
obj-m:=mydriver.o
module-objs:=mydriver.o
all:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
    rm -f *.ko
    rm -f *.o
```

(3) 使用 make 命令进行编译,并查看编译的结果,如图 8-22 所示。

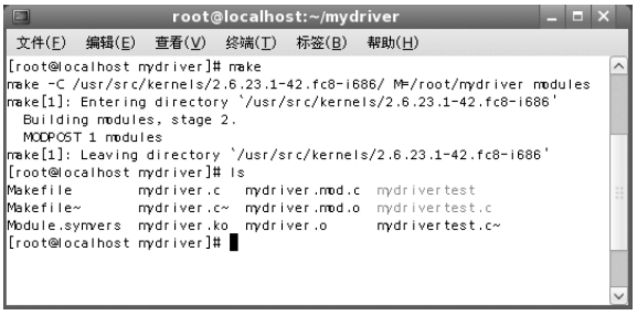


图 8-22 编译结果

图 8-22 中所示的 mydriver.ko 就是生成的驱动程序,其他文件为中间生成文件。

(4) 使用 insmod 命令加载模块,并使用 lsmod 命令查看加载结果,如图 8-23 所示。

(5) 建立设备节点,并使用 ls 命令进行查看,结果如图 8-24 所示。

(6) 编写测试程序 mydriver.c,源程序如下:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
```

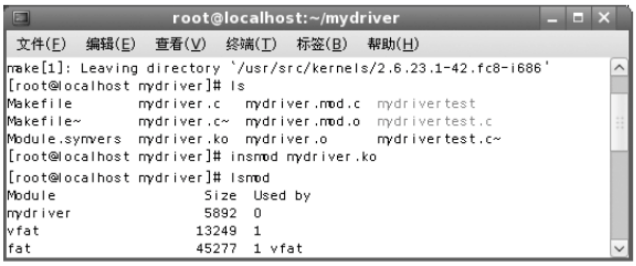


图 8-23　模块加载结果

图 8-24　建立设备节点

```
int main()
{
  int fd;
  int i;
  char buf[10];
  /* 调用 open 函数 * /
  fd=open("/dev/mydriver",O_RDWR);
  if(fd==-1)
  {
     printf("can't open file\n");
     exit(0);
  }
  /* 调用 read 函数 * /
  read (fd,buf,10);
  for(i=0;i<10;i++)   printf("%d",buf[i]);
  printf("\n");
  close(fd);
}
```

该程序调用了自定义的字符设备中的 open()函数和 read()函数。

(7) 编译并运行测试程序,可看到图 8-25 所示的结果。

图 8-25　测试程序运行结果

由于设计的字符设备 read()操作实现了从内核空间向用户空间分配 ASCII 码值为 1

的字符，所以测试程序运行结果中的一串字符的 ASCII 码值为 1。读者如果有兴趣，可以更改该设备定义中的 mydriver_read()操作使用的_put_user()函数，更改第一个参数的 ASCII 码值，重新编译执行并观察结果。

（8）卸载驱动模块并删除字符设备文件：

```
# rmmod mydriver
# rm /dev/mydriver
```

可以再次执行 lsmod 和 ls/dev/mydriver 命令，查看一下该字符设备是否还存在。

基于 的 远程控制系统设计

问题：设计一个基于 Web 的远程监控系统需要哪些步骤？

重点：LED 驱动程序的设计、表单设计和 CGI 程序设计。

内容：基于 Web 的远程监控系统设计的全过程，包括 LED 驱动程序设计、表单设计和相应的 CGI 程序设计。

现在设计一个简单的 LED 控制页面。当输入 1/0 时对应的 LED 灯亮/灭。LED 驱动程序使用 insmod 命令加载，CGI 程序编译后存放在 var/www/cgi-bin 目录下。

8.5.1 LED 驱动程序设计

任务：掌握 LED 控制寄存器的设置。理解驱动程序的内容。

在嵌入式系统的设计中，LED 一般直接由 CPU 的 GPIO(通用可编程 I/O)控制。本书中介绍的实验箱使用引脚 GPF4～GPF7。控制器一般由两组寄存器控制，即一组控制寄存器和一组数据寄存器。控制寄存器可设置 GPIO 口的工作方式为输入或输出。对 LED 的具体设置如下。

（1）引脚功能设为输出。

（2）要点亮 LED，令引脚输出 0。

（3）要熄灭 LED，令引脚输出 1。

1. 编写驱动程序

为了能够远程监控 LED，还需要编写 LED 驱动程序。下面是已经编写好的 LED 驱动程序，后面将按照函数调用的顺序进行讲解。

```
//s3c2440_leds.c
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/devfs_fs_kernel.h>
#include <linux/miscdevice.h>
#include <linux/delay.h>
```

```
#include <asm/irq.h>
#include <asm/arch/regs-gpio.h>
#include <asm/hardware.h>
#define DEVICE_NAME "leds"
#define LED_MAJOR 215
//设定 LED 使用的 GPIO 引脚
static unsigned long leds_table []={
S3C2410_GPF4,
S3C2410_GPF5,
S3C2410_GPF6,
S3C2410_GPF7,
};
//引脚功能设定为输出
static unsigned int leds_cfg_table []={
S3C2410_GPF4_OUTP,
S3C2410_GPF5_OUTP,
S3C2410_GPF6_OUTP,
S3C2410_GPF7_OUTP,
};
//应用程序对设备文件/dev/leds 执行 ioctl()函数时调用
static int s3c2440_leds_ioctl(struct inode * inode,struct file * file,unsigned int cmd,
unsigned long arg)
{
  switch(cmd)
  {
    case 0:
    case 1:
    if (arg >4) {
    return -EINVAL;
  }
//设置指定引脚的输出电平
s3c2410_gpio_setpin(led_table[arg], !cmd);
return 0;
default:
return -EINVAL;
  }
}

/* 这个结构是字符设备驱动程序的核心,当应用程序操作设备文件时会调用 open、read、write 等函数,最终会调用这个结构中的对应函数 */
static struct file_operations s3c2440_leds_fops=
{
  .owner=THIS_MODULE,/* 这是一个宏,指向编译模块时自动创建的_this_module 变量 */
  .ioctl=sbc2440_leds_ioctl,
};

//模块的初始化函数
static int __init sbc2440_leds_init(void)
{
  int ret;
  int i;
```

```
    /* 注册字符设备驱动程序,参数为主设备号、设备名字、file_operations 结构;这样,主设备号就
       和具体的 file_operations 结构联系起来了。操作主设备为 LED_MAJOR 的设备文件时,会调用
       sbc2440_leds_fops 中的相关成员函数,LED_MAJOR 也可以设为 0,表示由内核自动分配主设备
       号 */
    ret=register_chrdev(LED_MAJOR, DEVICE_NAME, &s3c2440_leds_fops);
    if (ret < 0)
    {
      printk(DEVICE_NAME " can't register major number\n");
      return ret;
    }
    /* 内核驱动注册完成后,要用以下代码创建设备文件 */
    devfs_mk_cdev(MKDEV(LED_MAJOR, 0), S_IFCHR | S_IRUSR | S_IWUSR | S_IRGRP,
    DEVICE_NAME);
    for (i=0; i < 4; i++)
    {
      s3c2410_leds_cfgpin(led_table[i], led_cfg_table[i]); //选择引脚的功能
      s3c2410_leds_setpin(led_table[i], 1); //设置指定引脚的输出电平为 1
    }
    printk(DEVICE_NAME " initialized\n");
    return 0;
  }

  //模块卸载函数
  static void_exit sbc2440_leds_exit(void)
  {
    devfs_remove(DEVICE_NAME); //内核驱动使用其移除设备文件
    unregister_chrdev(LED_MAJOR, DEVICE_NAME); //卸载驱动程序
  }

  /* 这两行指定驱动程序的初始化函数和卸载函数 */
  module_init(s3c2440_leds_init);
  module_exit(s3c2440_leds_exit);
```

执行 insmod sbc2440_leds. ko 命令时会调用 sbc2440_leds_init 函数,该函数再调用 register_chrdev 函数向内核注册驱动程序:将主设备号 LED_MAJOR 与 file_operations 结构 s3c2440_leds_fops 联系起来。之后应用程序操作主设备号为 LED_MAJOR 的设备文件时,比如 open、read、write、ioctl 等,s3c2440_leds_fops 中的相应成员函数就会被调用。但是,在 s3c2440_leds_fops 中并不需要实现所有这些函数,根据需要实现即可。

执行 rmmod s3c2440_leds. ko 命令时会调用 s3c2440_leds_exit 函数,该函数再调用 unregister_chrdev 函数卸载驱动程序,它的功能与 register_chrdev 相反。

s3c2440_leds_init 和 s3c2440_leds_exit 函数前的__init、__exit 只有在将驱动程序静态链接进内核时才有意义。前者表示 s3c2440_leds_init 函数的代码放在“. init. text”段中,这个段在使用一次后被释放(这样可以节省内存);后者表示 s3c2440_leds_exit 函数的代码被放在“. exit. data”段中,在链接内核时这个段没有使用,因为不可能卸载静态链接的驱动程序。

sbc2440_leds_fops 的组成如下:

```
static struct file_operations s3c2440_leds_fops={
```

```
.owner=THIS_MODULE,
.ioctl=s3c2440_leds_ioctl,
};
```

宏 THIS_MODULE 在 include/linux/module.h 中的定义如下：

```
#define THIS_MODULE(&__this_module)
```

__this_module 变量在编译模块时会自动创建，无须考虑。

file_operations 类型的 s3c2440_leds_fops 结构是驱动程序中最重要的数据结构，编写字符设备驱动程序的主要工作也是填充其中的各个成员。

应用程序执行系统调用 ioctl(fd,cmd,arg)时，s3c2440_leds_ioctl 函数将被调用。该函数中调用的 s3c2410_gpio_setpin 函数是在内核中实现的，它通过 GPIO 的数据寄存器来设置引脚的输出电平：输出 0 时点亮 LED，输出 1 时熄灭 LED。

2．编写 Makefile 文件

为了编译驱动程序，需要编写 Makefile 文件，其内容如下：

```
obj-m +=s3c2440_leds.o
KDIR:=/lib/modules/2.6.13/build
all:
make -C $(KDIR) M=$(PWD) modules
clean:
make -C $(KDIR) M=$(PWD) clean
```

执行 make 命令，生成驱动程序 s3c2440_leds.ko，此时便可以使用 insmod s3c2440_leds.ko 和 rmmod s3c2440_leds.ko 动态加载和卸载驱动程序了。

8.5.2　表单设计

设计表单 led.html，通过表单，接收用户输入的数据，并调用相应的 CGI 程序，完成对 LED 的远程监控。其内容如下：

```
<html>
<title>LED 远程监控</title>
<body>
<form action="/cgi-bin/cgi_led.cgi" method=GET>
<font size=7><center>基于 s3c2440 的 Web 服务器的设计程序</center></font><br><p><p>
<center>系统资源：s3c2440,16MB Flash,32MB SDRM,IP:192.168.2.23</center><P>
<center>输入要点亮的 LED:
<input type=text name="led" ></center><br>
<center>输入 LED 状态：
<input type=text name="status" ></center><br>
<center><input type=submit value="确定"><input type=reset value="重设"></center>
</form>
</body>
</html>
```

运行该网页，结果如图 8-26 所示。

当单击“确定”按钮后，便会将控制数据传递给 CGI 程序，完成对 LED 的远程监控。

图 8-26　LED 远程监控页面

8.5.3　CGI 程序的编写

编写 CGI 程序：cgi_led.c，主要功能是当接收页面传过来的 0 或 1 时，点亮或熄灭相应的 LED 灯，内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#define DEVICE_GPIODRV "/dev/gpios"
int main()
{
  int fd;
  int led;
  int status;
  char * data;
  if((fd=open(DEVICE_GPIODRV,O_RDONLY | O_NONBLOCK))<0)
  {
    printf("open device: %s\n",DEVICE_GPIODRV);
    perror("can not open device");
    exit(1);
  }
  printf("Content-type: text/html;charset=gb2312\n\n");
  printf("<html>\n");
  printf("<head><title>CGI LED DEMO</title></head>\n");
  printf("<body>\n");
  printf("<h1>CGI LED DEMO 1:0 led1 on 1:1 led1 off</h1>\n");
  printf("</body>\n");
```

```
    data=getenv("QUERY_STRING");
    if(sscanf(data,"led=%ld&status=%ld",&led,&status)!=2)
    {
      printf("<p>请正确输入");
      printf("</p>");
    }
    if(led>3)
    {
      printf("<p>Please input 0<=led<=3!");
      printf("</p>");
    }
    if(status>1)
    {
      printf("<p>Please input 0<=status<=1!");
      printf("</p>");
    }
    ioctl(fd,status,led);
    close(fd);
    printf("</html>\n");
    exit(0);
}
```

至此便完成了整个基于 Web 的远程监控系统的设计。整个过程可以总结如下。

(1) 编写 LED 的驱动程序,以便能够驱动相应的 LED。

(2) 编写 HTML 表单,将用户的控制数据提交给服务器上的 CGI 程序。

(3) 编写 CGI 程序,处理表单提交的控制数据,并实现对 LED 的远程控制。

小结

本章详细介绍了一个基于 Web 的远程监控系统的设计过程,包括系统架构设计和软硬件的实现。主要内容如下。

1. 基于嵌入式 Web 的远程监控系统简介

主要介绍了嵌入式 Web 服务器和远程监控系统的概念,以及几个典型的嵌入式 Web 的远程监控系统的应用。通过典型应用使学生对嵌入式 Web 远程监控系统有了基本的认识。

2. 系统架构设计

介绍了一个嵌入式 Web 远程监控系统的整体架构设计,其中网络架构确定以嵌入式 Web 服务器为中心,通过 Internet 远程访问嵌入式 Web 服务器,嵌入式 Web 服务器通过现场总线控制各个节点,以达到远程监控的目的;硬件架构采用 Samsung 公司的主流 ARM9 处理器 S3C2440A 进行构建;软件架构采用 Linux 作为操作系统平台,选择 boa+CGI 方案。

3. 系统硬件实现

直接采用上海双实科技有限公司的 SinoSys-M31 嵌入式教学实验箱。

4. 系统软件实现

这是整个系统的重点,主要包括嵌入式 Web 服务器 boa 的移植和配置、HTML 中表单

的使用和 CGI 程序设计。

5. Linux 设备驱动程序设计

介绍了 Linux 驱动程序的组成和设计方法，以及一个典型的字符设备驱动程序的开发、编译、加载和卸载的全过程。

6. 基于 Web 的 LED 远程监控系统设计

介绍了基于 Web 的 LED 远程监控系统设计的全过程，包括 LED 驱动程序设计、表单设计和相应的 CGI 程序设计。

参考文献

[1] 刘洪涛,孙天泽.嵌入式系统技术与设计[M].北京:人民邮电出版社,2008.

[2] 孙秋野,孙凯,冯健.ARM嵌入式系统开发典型模块[M].北京:人民邮电出版社,2007.

[3] 符意德,陆阳.嵌入式系统原理及接口技术[M].北京:清华大学出版社,2007.

[4] 赵苍明,穆煜.嵌入式Linux应用开发教程[M].北京:人民邮电出版社,2009.

[5] 宋宝华.Linux设备驱动程序开发详解[M].北京:人民邮电出版社,2008.

[6] 孙弋.ARM-Linux嵌入式系统开发基础[M].西安:西安电子科技大学出版社,2008.

[7] 王进德.嵌入式Linux程序设计与应用案例[M].北京:中国电力出版社,2007.

[8] 陈渝,韩超,李明.嵌入式系统原理及应用开发[M].北京:机械工业出版社,2008.

[9] 上海双实科技有限公司.ARM嵌入式系统设计及接口编程实验教程[M].2007.

[10] 上海双实科技有限公司.ARM/Linux嵌入式系统实验教程[M].2007.

[11] 汤晓丹,梁红兵等.现代操作系统[M].北京:电子工业出版社,2008.

[12] 任哲,潘树林,房红征.嵌入式操作系统基础UC/OS-Ⅱ和Linux[M].北京:北京航空航天大学出版社,2006.